SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

Matija Karašić

RJEŠAVANJE FIZIKALNIH PROBLEMA POMOĆU PROGRAMSKOG JEZIKA JULIA

Diplomski rad

Zagreb, 2016.

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

SMJER: NASTAVNIK FIZIKE

Matija Karašić

Diplomski rad

Rješavanje fizikalnih problema pomoću programskog jezika Julia

Voditelj diplomskog rada: doc. dr. sc. Davor Horvatić

Ocjena diplomskog rada: _____

Povjerenstvo: 1. _____

2. _____

3. _____

Datum polaganja: _____

Zagreb, 2016.

ZAHVALJUJEM:

Mentoru, doc. dr. sc. Davoru Horvatiću, na pomoći pri odabiru teme diplomskog rada, strpljenju i pažnji kojom je pratio sve faze mog rada, te korisnim savjetima i nesebičnom prenošenju znanja.

Dr. sc. Maji Planinić na konstruktivnim savjetima pri izradi metodičkog dijela rada.

Sestri Ivani na strpljenju i pomoći pri ispravljanju rada.

Posebnu zahvalnost iskazujem cijeloj svojoj obitelji koja me podržava tijekom školovanja.

Sažetak

U ovom radu je dan kratak pregled programskog jezika Julija i njegova mogućnost rješavanja fizikalnih problema, kroz probleme vizualizacije podataka, obrade rezultat mjerenja, gibanja gušenog harmoničkog oscilatora i gibanja čestice u potencijalu. Dobivena numerička rješenja dana su u grafičkom obliku, gdje je to bilo moguće. Metodički dio čini učenički projekt kojem je cilj povezivanje znanja fizike i informatike kroz numeričku integraciju kinematičkih veličina.

Solving Physics Problems Using Julia Programming Language

Abstract

This thesis gives short overview of Julia programming language and its abilities for solving physical problems, data visualization, and processing measurement results. Motion of damped harmonic oscillator and motion of particle for specified potential is discussed in detail. Obtained numerical solutions are given in graphic form, wherever possible. Educational part is composed of a student project with goal of linking knowledge of physics and computer science, through numeric integration of kinematic quantities.

Sadržaj

1 Uvod

Tema ovog rada je pregled programskog jezika Julia, koji se želi nametnuti kao *de facto* alat za numeriku u znanosti. Tema je obrađena kroz kratki pregled funkcinalnosti Julije i nekoliko fizikalnih problema riješenih numeričkim metodama koristeći Juliju i njezine pakete.

Prvi dio rada je kratki uvod o Juliji i usporedba s već postojećim rješenjima slične namjene, kao što su: Python, R, Matlab i Mathematica. Također se obrađuje Julijino osnovno sučelje REPL i korištenje dodatnih paketa pisanih za Juliju. Dalje je dan pregled Julije i njezine sintakse kroz varijable, funkcije, kontrolu toka i metode za unos i spis podataka. Kraj prvog dijela rada je zaključen pregledom mogućnosti za pozivanje vanjskih funkcija.

Drugi dio rada čine problemi kroz koje su demonstrirane Julijine mogućnosti, od onih jednostavnijih do kompliciranijih koji zahtijevaju korištenje specijaliziranih paketa i algoritama. Problemi koji su obrađeni su vizualizacija podataka, obrada rezultata mjerenja, gušeni harmonički oscilator i gibanje čestice u potencijalu.

Završni dio rada čini metodički dio. On se sastoji od kratkog uvoda u kojem se navode razne poteškoće koje učenici imaju u razumijevanju grafova kinematičkih veličina, te se opisuje projekt koji bi učenicima mogao približiti te veličine, ali i povezati znanja koja su stekli na nastavi informatike sa znanjima koja su stekli na nastavi fizike.

2 Programski jezik Julia

2.1 Osnovni podaci o programskom jeziku Julia

Julia je programski jezik koji se posljednjih godina razvija na Massachusetts Institute of Technology (MIT). Prvi put je objavljen u veljači 2012. godine. Njegovu jezgru, kao i velik broj dodatnih paketa, razvili su Jeff Bezanson, Stefan Karpinski, Viral Shah i Alan Edelman. Njihova želja za novim programskim jezikom je proizašla iz činjenica da su većina alata, koje su koristili, imali uz svoje prednosti i očite nedostatke. Dok je jedne krasila brzina, drugi su imali jasnoću i jednostavnost korištenja. Najveći nedostatak je bio upravo činjenica da nisu imali jedinstveni alat s kojim bi radili, nego više njih.

Juliu karakteriziraju: mogućnost definiranja funkcija za različite tipove podataka (multiple dispatch), sustav dinamičkih tipova podataka, LLVM JIT (low-level virtual machime just in time) prevoditelj, ugrađeni upravitelj paketa, makro naredbe, mogućnost pozivanja Python i C funkcija, moćne shell mogućnosti i upravljanje drugim procesima, paralelno i distribuirano računalstvo, mogućnost definiranja tipova podataka, elegantna konverzija i promocija numeričkih i ostalih tipova podataka, podrška za Unicode standard i MIT licenca.

Multiple dispatch je sustav koji omogućava definiranje funkcije s istim imenom, a koja će se znati nositi s različitim tipovima podataka koje dobiva kao ulazne varijable. Takav sustav u kombinaciji s mogućnošću definiranja vlastitih tipova podataka, uz ostale karakteristike upravljanja podacima, makro naredbe i sintaksu sličnu Pythonovoj, dao je Juliji željenu jednostanost. A JIT kompajler (prevođenje se događa dok se program izvodi) kombinira brzinu izvođenja prevedenog koda i fleksibilnost interpretera, čime se dobiva brzina bliska onoj programa napisanih u C-u. Julia također podržava paralelno računalstvo, bilo na više jezgri na jednom procesoru ili distribuirano na mreži na više računala.

Za Juliu je još bitno naglasiti da je vrlo jednostavna za učenje, s obzirom na sličnost sintakse Pythonu, ali i ostalim jezicima koji su joj uzor, kao što je Matlab. Također sama jezgra Julije je licencirana MIT-jevom licencom, što znači da je besplatna i otvorenog koda, čime se potiče njezin daljnji razvoj.

Julia je dostupna na sve tri veće platforme: Windowsima, Linuxu i MacOS-u. Dolazi u x86 i x64 verzijama. Moguće ju je preuzati kao arhivu ili instalacijski paket. Na Linuxu je dostupna u svim većim repozitorijima. Također je moguće i preuzeti izvorni kod i izgraditi ju iz njega.

2.1.1 Usporedba s drugim programskim jezicima

Julijina glavna prednost pred konkurentskim jezicima kao što su Python, R i Matlab je brzina. Testiranja koje su obavili sami razvijatelji Julije za različite algoritme pokazuju relativnu brzinu u odnosu na C (brzina izvođenja u C-u je 1.0). Ovi testovi

	Julia	Python	R	Matlab	Mathematica
fib	2.11	77.76	533.52	26.89	118.53
parse_int	1.45	17.02	45.73	802.52	15.02
quicksort	1.15	32.89	264.54	4.92	43.23
mandel	0.79	15.32	53.16	7.58	5.13
pi_sum	1.00	21.99	9.56	1.0	1.69
rand_mat_stat	1.66	17.93	14.56	14.52	5.95
rand_mat_mul	1.02	1.14	1.57	1.12	1.30

Tablica 2.1: Usporedba brzine izvođenje raznih algoritama u Juliji, Pythonu, R-u, Matlabu i Mathematici s C-om [4]

su svi izvođeni na istom računalu, bez podrške za paralelno računalstvo, te je svaki jezik izvodio isti algoritam. Julia se u svim testovima pokazala kao usporediva, a u jednom čak i brža od C-a. Ostali jezici su u većini slučajeva bili višestruko sporiji i od C-a i od Julije.

Julija se pokazala bržom od Pythona koji je postao gotovo standardan jezik u znanosti zahvaljujući svojim modulima NumPy i SciPy, te velikom broju korisnika, kao i činjenici da je kao i Julia jezik otvorenog koda. Julia je po sintaksi slična Python-u, što uvelike olakšava prelazak s Pythona, s kojim se susreo gotovo svaki moderni korisnik koji zna programirati, a više neće biti potrebe prepisivati kod napisan u Pythonu u C da bi se brže izvodio.

R je jezik otvorenog koda koji se najčešće koristio za statističku obradu podataka i vizualizaciju. Vrlo je spor jer se izvodi na jednoj jezgri i paralelizacija nije jednostavna.

Matlab je alat za operacije s matricama. Julija ima sličnu sintaksu, ali se svejedno pokazala bržom u linearnoj algebri za koju se inače upotrebljava Matlab.

Mathematica je alat za simoličko rješavanje matematičkih problema, sa sučeljem i sintaksom koja gotovo sliči zapisu na papiru. Također ima brojne alate za vizualizaciju, ali što se same brzine tiče ne može se nositi s Julijom.

2.2 REPL

REPL je skraćenica od read-evaluate-print-loop. To je osnovno sučelje Julije, to jest njezin naredbeni redak. U REPL se može izravno unositi kod koji će tada biti izvršen pritiskom na tipku `Enter`. Ako unesemo niz znakova, Julia će jednostavno vratiti taj isti niz znakova, a ako unesemo neki izraz Julia će izvršiti izraz i vratiti nam rezultat:

```
julia> "Hello!"
"Hello!"
```

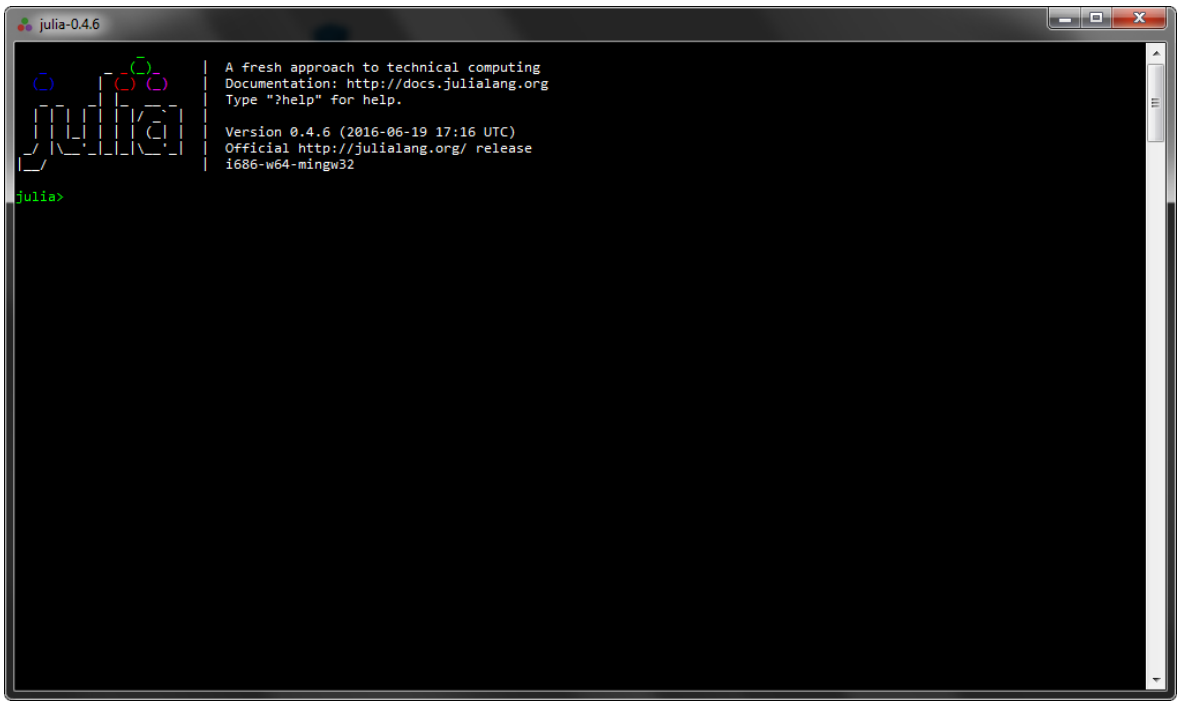
```
julia> 2+3
5
```

Ako ne želimo da Julija vrati rezultat ili želimo u istom redu zadati više naredbi koristimo *;*. Za poziv zadnjeg rezultata koristimo varijablu *ans* (ova varijabla postoji samo u REPL-u), a varijabli zadajemo vrijednost znakom jednakosti =:

```
julia> a=2; b=3; c=a+b;

julia> c
5

julia> 2ans
10
```

U prošlom primjeru vidimo da množenje pomoću operatora * nije potrebno eksplicitno navesti, ali ga je u praksi bolje koristiti, jer dva znaka jedan do drugog Julia shvaća kao novu varijablu:

```
julia> a*b
6

julia> ab
ERROR: UndefVarError: ab not defined
```



Slika 2.1: Julia REPL u Windows operativnom sustavu

Neke naredbe moraju biti napisane u više redova. Kada REPL prepozna takvu naredbu pritisak na tipku `Enter` će prebaciti kursor u novi red. Tipka `Enter` će izvršiti naredbu tek kada REPL prepozna da je unutarnja struktura naredbe sintaktički zatvorena.

```
julia> if a>b

         println("a($a) je vece od b($b)")

       else

         println("a($a) nije vece od b($b)")

       end
a(2) nije vece od b(3)
```

Ovaj primjer je koristio funkciju *println()*, koja kao argument prima niz znakova koji onda ispisuje kao rezultat i prelazi u novi red. Ako u nizu znakova želimo ispisati vrijednost varijable koristimo simbol *$* i ime varijable. Uz funkciju *println()* postoji i funkcija *print()* koja nakon ispisa ne prelazi u novi red. Julia također pomoću REPL-a ima implementiran sustav pomoći. Sustavu pomoći se pristupa tako da se upiše znak *?* koji mijenja naredbeni red u upit za pomoć u koji tada upisujemo ime funkcije, tipa, makro naredbe ili slično.

```
help?> +
search: + .+

  +(x, y...)

  Addition operator. x+y+z+... calls this function with all
  arguments, i.e. +(x, y, z, ...).
```

Iako je moguće raditi samo pomoću REPL-a u većini slučajeva to nije praktično. Željeli bismo imati skriptu koju je moguće spremiti za kasniju upotrebu. Takvu skriptu je moguće napisati u bilo koje uređivač tekstualnih datoteka (spremiti s ekstenzijom .jl) i kasnije je izvoditi iz REPL-a.

```
julia> include("hi.jl")
Hello, World!

julia> c
15
```

Pomoću funkcije *include()* smo izvršili skriptu naziva hi.jl. Skripta se sastojala

samo od dvije linije koda:

```
c=15
println("Hello, World!")
```

Vidimo da skripta definira varijablu *c*, ali je ispisan samo rezultat zadnje naredbe u skripti.

Za rad u REPL-u je još bitno istaknuti da se za izlazak koristi funkcija *quit()* ili kratica na tipkovnici `Ctrl+D`.

2.3 Korištenje paketa

Julija osim standardnih funkcija i naredbi ima i veliki broj dodatnih paketa. Oni značajno proširuju Julijinu funkcionalnost. Za njihovu instalaciju u REPL je ugrađen upravitelj paketa. Funkcijom *Pkg.status()* se dobiva popis instaliranih paketa i njihova verzija. Funkcija *Pkg.update()* ažurira instalirane pakete.

Za instalaciju novih paketa se koristi funkcija *Pkg.add()*. Kao argument prima ime paketa koji instalira. Ako se paket za pravilan rad oslanja na neke druge pakete i oni će biti instalirani.

```
julia> Pkg.add("ASCIIPlots")
```

Da bi se paket koristi prvo je potrebno naredbom *using* učitati paket. Nakon toga je moguće koristiti funkcije iz paketa.

```
julia> using ASCIIPlots

julia> x=0.0:0.01:2pi; y=sin(x); lineplot(x, y)
```

Paketi, koji više nisu potrebni, se uklanjaju funkcijom *Pkg.rm()*.

2.4 Varijable i tipovi podataka

Sustavi tipova se dijele na dvije vrste: statički i dinamički. Kod statičkih tipova svaki izraz mora imati definirani tip prije izvršenja programa, dok se kod dinamičkih tipova ništa ne zna o tipu sve do samog izvođenje programa.

Julija koristi sustav dinamičkih tipova podataka, a to znači da je Julija programski jezik kod kojeg nije potrebno posebno naznačiti tip podatka kada deklariramo varijablu, nego će sama odraditi tip. Svaki izraz u Juliji ima tip i s obzirom na to Julija spada u čvrsto-tipizirane jezike. Ali Julijin sustav tipova je takav da iskorištava neke prednosti statičkih sustava tipova. Tako je moguće naznačiti tip za neke vrijednosti. Kada se to ne učini dopuštamo da vrijednost bude bilo kojeg tipa. Opcija da Julija sama odredi tip varijable je značajka koja omogućava lakše učenje jezika početnicima, ali mogućnost da sami odredimo tip je značajka koja će Juliji dati nje-

zinu brzinu. Kada sami zadamo tip varijable Julija ga ne mora sama odrađivati, a s obzirom da je sve u Juliji izraz, to jest ima tip, program koji ima optimalno odrađene tipove će se interpretirati brže. Najčešće se tipovi određuju u deklaracijama funkcijama, to omogućava korištenje multipe dispatcha, koji je jedna od glavnih značajki Julije.

Apstraktni tipovi se ne daju instancirati, a služe za opisivanje i stvaranje hijerarhije konkretnim tipovima. Oni čine osnovu Julijinog sustava tipova. Tip *Any* se nalazi na vrhu te hijerarhije. Ako deklariramo funkciju, a ne zadamo tip vrijednosti, one će biti tipa *Any*. Još neki apstraktni tipovi su *Number*, *Real*, *AbstractFloat*, *Integer*, *Signed*, *Unsigned*. Suprotnost tipa *Any* se zapisuje kao *Union*{}. Pomoću funkcije *i:* možemo provjeriti je li neki tip potomak supertipa, gdje je supertip onaj koji se u hijerarhiji nalazi iznad njega.

```
Number <: Any              #daje rezultat true
Real <: Number             #daje rezultat true
AbstractFloat <: Real      #daje rezultat true
Integer <: Real            #daje rezultat true
```

Vidi se da je *Number* potomak tipa *Any*, a da je njegov potomak tip *Real*, čiji su potomci tipovi *AbstractFloat* i *Integer*. Imena tipova se pišu velikim početnim slovom, a ako se ime sastoji od dvije riječi, svaka se piše velikim slovom, ali bez razmaka, na primjer *AbstractFloat* ili *BigInt*.

Varijabla je ime pridruženo nekoj vrijednosti. U Juliji se varijabli zadaje vrijednost znakom pridruživanja =, a tip se može provjeriti funkcijom *typeof()* koja vraća ime tipa.

```
julia> x=7; typeof(x)
Int32

julia> x=7.0; typeof(x)
Float64
```

2.4.1 Cijeli brojevi

Cijeli broj se obično naziva integer (engl. integer = cijeli broj). U Juliji cijeli brojevi su predstavljeni tipovima *Int8*, *Int16*, *Int32* i *Int128*. Osim ovih tipova možemo i koristiti tipove bez predznaka, znači samo pozitivni brojevi: *UInt8*, *UInt16*, *UInt32* i *UInt128*. U Juliji su dostupne funkcije koje vraćaju najmanju i najveću vrijednost za određeni tip. To su funkcije *typemin()*, odnosno *typemax()*.

```
julia> a=typemin(Int32); b=typemax(Int32); [a,b]
2-element Array{Int32,1}:
 -2147483648
```

Tip	Predznak	Broj bitova	Minimum	Maksimum
Int8	✓	8	-2^7	2^7-1
UInt8		8	0	2^8-1
Int16	✓	16	-2^{15}	$2^{15}-1$
UInt16		16	0	$2^{16}-1$
Int32	✓	32	-2^{31}	$2^{31}-1$
UInt32		32	0	$2^{32}-1$
Int64	✓	64	-2^{63}	$2^{63}-1$
UInt64		64	0	$2^{64}-1$
Int128	✓	128	-2^{127}	$2^{127}-1$
UInt128		128	0	$2^{128}-1$

Tablica 2.2: Pregled tipova koji predstavljaju cijele brojeve [4]

```
2147483647
```

Cijeli brojevi bez predznaka mogu se unositi u binarnom, oktalnom i heksadekadskom obliku. Za to se koriste predznaci *0b*, *0o*, odnosno *0x*. Julija ih ispisuje u heksadekadskom obliku.

```
julia> 0b1110
0x0e

julia> typeof(ans)
UInt8
```

Za slučajeve u kojima je potrebna veća duljina od 128 bita postoji tip koji koristi proizvoljnu duljinu *BigInt*.

```
julia> x=BigInt(2); x^200>typemax(Int128)
true
```

2.4.2 Realni brojevi

Realni brojevi u Juliji su dostupni u poloviČnoj, jednostrukoj i dvostrukoj preciznosti. Predstavljeni su tipovima *Float16*, *Flost32*, odnosno *Float64*. Obično se nazivaju float (engl. floating point number = broj s pomičnim zarezom). Kao i za cijele brojeve postoji tip *BigFloat* koji omogućava računanje u proizvoljnoj preciznosti.

Za realne brojeve postoji i zapis u znanstvenoj notaciji gdje se koristi *e* za dvostruku preciznost odnosno *f* za jednostruku preciznost.

```
julia> 1.5e-2
0.015
```

```
julia> 1.5f-2
0.015f-2
```

Uz brojeve su definirane i posebne vrijednosti u radu s realnim brojevima. To su negativna i pozitivna beskonačnost i vrijednost koja označava da nije broj. Negativna beskonačnost (*-Inf16*, *-Inf32*, *-Inf*) je manja od svih konačnih realnih brojeva. Pozitivna beskonačnost (*Inf16*, *Inf32*, *Inf*) je veća od svih konačnih realnih brojeva. Vrijednost koja nije broj *NaN* (engl. not a number = nije broj) je različita od svih realnih brojeva, pa i same sebe. S tim vrijednostima je moguće i računati.

```
julia> 1.0/Inf
0

julia> 1.0/0.0
Inf

julia> 100-Inf
-Inf

julia> Inf-Inf
NaN
```

2.4.3 Racionalni i kompleksni brojevi

Julija ima mogućnost definiranja racionalnih i kompleksnih brojeva. To su parametarski tipovi, gdje su racionalni brojevi tipa $Rational\{T\}$, a kompleskni tipa $Complex\{T\}$. T može biti tip cijelih ili realnih brojeva. Racionalni brojevi se definiraju pomoću simbola *//*, a kompleksni pomoći imaginarne jedinice *im*.

```
julia> a=2//3; b=1.3+2.4im; [typeof(a), typeof(b)]
2-element Array{DataType,1}:
 Rational{Int32}
 Complex{Float63}
```

Svi ovi tipovi se mogu koristiti u operacijama s drugim tipovima brojeva bez da

Tip	Preciznost	Broj bitova
Float16	polovična (half)	16
Float32	jednostruka (single)	32
Float64	dvostruka (double)	64

Tablica 2.3: Pregled tipova koji predstavljaju realne brojeve [4]

Julija prijavi ikakvu grešku.

```
julia> a+b
1.9666666666666668 + 2.4im
```

U radu s racionalnim brojevima se može koristiti funkcija koja ga pretvara u realan broj *float()*, a definirane su u funkcije *num()* i *den()* koje kao rezultat vraćaju brojnik, odnosno nazivnik.

```
julia> float(a)
0.6666666666666666

julia> num(a)
2

julia> den(a)
3
```

2.4.4 Logički tipovi

Julia ima definiran logički tip *Bool*. On će najčešće biti tip rezultat usporedbe i logičkih operacija. Ima dvije moguće vrijednosti *true* (istinito) i *false* (neistinito). U nekim programskim jezicima te vrijednosti predstavljaju jedinicu i nulu. To u Juliji nije slučaj iako postoje funkcije u kojima dolazi do promocije tipa *Bool* u cjelobrojni tip *Int32*.

```
julia> 1<2
true

julia> typeof(ans)
Bool

julia> true+true
2

julia> typeof(ans)
Int32
```

2.4.5 Nizovi podataka

Kada je potreban interval brojeva moguće ga je definirati pomoću tipa $UnitRange\{T\}$. Koristimo znak *:*, tako da prvo navedemo broj od kojeg želimo početi, a zatim onaj s kojim želimo završiti. Između njih je moguće navesti veličinu koraka, a ako nije

navedena onda je 1.

```
julia> a=1:10; b=0:0.25:1;
```

Iteracija po varijabli a bi dala brojeve 1, 2, 3, 4, 5, 6, 7, 8, 9 i 10, a po varijabli b 0.0, 0.25, 0.5, 0.75 i 1.0. Ovakva konstrukcija je korisna za korištenje u raznim vrstama petlji.

Također je moguće konstruirati i linearne ili logaritamske nizove u kojima je određen broj elemenata, a ne korak koji koristimo. Njih tvorimo pomoću funkcija *linspace()*, odnosno *logspace()*. Obje funkcije uzimaju tri argumenta. Kod funkcije *linspace()* prva dva argumenta su početak i kraj intervala. Kod funkcije *logspace()* to su potencije broja 10 koje označavaju početak i kraj intervala. Kod obje funkcije zadnji argument je broj elemenata niza.

```
julia> a=linspace(0,1,5); b=logspace(0,3,4);
```

Sada je u varijabli a spremljen niz 0.0, 0.25, 0.5, 0.75, 1.0, a u varijabli b 1.0, 10.0, 100.0, i 1000.0.

U zadnjem primjeru tip podatka u varijabli a je *LinSpace{Float64}*, a u varijabli b to je *Array{Float64,1}*. Upravo je *Array* najčešće korišteni općeniti tip nizova podataka u Juliji. Array ili niz je indeksirani niz podataka različitih tipova homogenih podataka. To znači da neki niz može sadržavati vrijednosti tipa *Int32*, *Float64*, *Bool* ili druge, ali su sve vrijednosti u njemu istog tipa. Može se definirati na više načina.

```
arr1=[]              #stvara prazan array elemenata tipa Any
arr2=Float64[]       #stvara prazan array elemenata tipa Float64
arr3=Array(Int32,5)  #stvara array od 5 elemenata tipa Int32
arr4=["ab","cd"]     #stvara array od 2 elementa tipa String
arr5=collect(1:7)    #od intervala stvara niz elemenata tipa Int32
```

Za pristup određenom elementu koristi se ime varijable i redni broj elementa kojem se pristupa. Indeksiranje u Juliji počinje od broja 1, za razliku od većine drugih jezika kod koji počinje od 0.

```
julia> arr5[2]
2
```

Na isti način možemo i mijenjati pojedine elemente.

```
arr5[2]=10      #rezultat je [1,10,3,4,5,6,7]
arr5[5:end]=10 #rezultat je [1,10,3,4,10,10,10]
```

Za rad s nizovima možemo koristiti i posebne funkcije: *push!()*, *fill!()*, *pop!()* i *splice!()*. Funkcija *push!()* prima kao prvi argument ime niza, a drugi je element koji se dodaje nizu. Ova funkcija ne stvara novu varijablu, već mijenja postojeću dodajući joj element. Funkcija *fill!()* popunjava željeni niz nekom vrijednošću. Funkcija *pop!()*

kao rezultat vraća zadnji element niza i uklanja ga iz njega. Funkcija *splice!()* uklanja element s određenog mjesta u nizu.

```
push!(arr4,"ef") #rezultat je ["ab","cd","ef"]
fill!(arr4,"aa") #rezultat je ["aa","aa","aa"]
a=pop!(arr4)     #rezultat je a="aa", a arr4 je ["aa","aa"]
splice!(arr5,2)  #rezultat je [1,3,4,10,10,10]
```

Dužinu niza možemo saznati koristeći funkciju *length()* koja vraća broj elemenata.

```
julia> length(arr4)
2
```

2.4.6 Matrice

Matrice u Juliji su također tipa *Array*. Matrice se definiraju na sličan način kao i nizovi. U slučaju da se želi dobiti matrica upisuju se brojevi koji se odvajaju razmakom, a ne zarezom. Za novi red se koristi točka-zarez. Time se dobiva dvodimenzionalna struktura podataka.

```
julia> mat=[1 2;3 4]
2x2 Array{Int32,2}:
 1 2
 3 4
```

Elementima matrice se pristupa na sličan način kao i elementima niza. Da bi se pristupilo određenom elementu navodimo ime varijable i nakon nje indeks reda, a zatim indeks stupca. Također je moguć isti pristup kao i kod jednodimenzionalnih nizova gdje se navodi samo jedan broj nakon varijable. Tada se elementi matrice indeksiraju po stupcima. Prvo se indeksiraju elementi prvog stupca, zatim drugog i tako dalje.

```
julia> mat[2,1]
3


julia> mat[3]
2
```

2.4.7 Znakovni nizovi

Znakovni nizovi u Juliji su načešće jedan od dva tipa: *ASCIIString* ili *UTF8String*. String (engl. string = nit) je niz znakova (vrijednosti tipa *Char*). Bit će *UTF8String* ako sadrži znakove koji su definirani Unicode standardom, a nemaju ekvivalent u

ASCII standardu. Niz znakova se definira navodnim znacima.

```
julia> s1="abcd"
"abcd"
```

Elementima niza znakova se može pristupati na isti način kao i elementima niza. S iznimkom da se pojedini elementi ne mogu mijenjati jednom kada je niz znakova definiran (iako možemo cijeli niz znakova definirati ispočetka).

```
julia> s1[2]
'b'


julia> s1[2]="r"
ERROR: MethodError: `setindex!` has no method matching
setindex!(::ASCIIString, ::ASCIIString, ::Int32)
```

Također je moguće koristiti funkcije *length()* i *join()*. Prva funkcija, kao i prije, vraća dužinu niza znakova, a druga služi za spajanje više niza znakova. Ona prima kao argument niz nizova znakova koje onda vraća kao spojeni niz znakova.

```
length(s1)        #rezultat je 4
join([s1,"efgh"]) #rezultat je "abcdefgh"
```

Funkcija *join()* ima i opcionalni argument koji je isto niz znakova kojim se odvajaju nizovi znakova koji se spajaju.

```
julia> s2="Matija"; s3="Ivana"; join([s2,s3],", ")
```

Julia u svojoj sintaksi ima ograničeno formatiranje brojeva i nizova znakova. Najčešće se koristi simbol *$* da bi se vrijednost varijable pretvorila u niz znakova, ali s jako malo kontrole nad formatom (osim standardnih specijalni znakova kao što su: *\n*, *\t* *\r* i slični). Makro-naredba *printf()* sadrži opcije za formatiranje slične onima u C-u. Formatirani niz znakova se zapisuje sa simbolom na mjestu kojem će se zapisati vrijednost varijable.

```
julia> @printf("Autor ovog rada je %s \n", s2)
"Autor ovog rada je Matija"
```

Osim simbola *%s* koji se koristi za formatiranje niza znakova, postoje još simboli: *%d* za cijele brojeve, *%f* za realne brojeve, *e* za brojeve u znanstvenoj notaciji i *%c* za znakove.

```
x1=3.5667; x2=52.4; @printf("%10s \n%10.2f \n%10.1e", s2, x1, x2)
#>     Matija
#>       3.57
#>    5.2e+01
```

2.4.8 Definiranje složenih tipova

Osim tipova koji već postoje u Juliji moguće je definirati i vlastite tipove. Oni se mogu sastojati od više polja. Definiramo ih ključnim riječi *type*. Svakom polju se može specificirati tip, a ako se to ne učini bit će tipa *Any*.

```
type Inventar
  ID::Int32
  opis::ASCIIString
  cijena::Float64
  dostupnost::Bool
```

Nakon što je tip definiran možemo ga koristiti.

```
alien=Inventar(015080025, "i7 6700HQ, 8GB, GTX970M", 16299.90, false)
```

Moguće je pristupati pojedinim poljma složenog tipa, kao i mijenjati ih.

```
julia> alien.opis
"i7 6700HQ, 8GB, GTX970M"

julia> alien.dostupnost=true
true
```

2.5 Funkcije

Funkcije u Juliji su objekti koji primaju argumente, zatim na njima obavljaju neke operacije i vraćaju rezultat. Rezultat mogu biti jedna, više ili nijedna vrijednost. S obzirom na to da su funkcije u Juliji objekti prve klase, mogu biti proslijeđene kao argument drugim funkcijama, vraćene kao rezultat drugih funkcija ili mapirane na set vrijednosti.

2.5.1 Operatori i elementarne funkcije

Skoro svi operatori u Juliji koji rade s jednostavnim numeričkim tipovima su funkcije. Razlikuju se samo posebnom sintaksom, to jest mogu se pisati u interfiksnom obliku.

```
julia> 2+3+4
9

julia> +(2,3,4)
9
```

U elementarne funkcije koje možemo pisati kao operatore spadaju: zbrajanje (+, funkcija *+()*), oduzimanje (-, funkcija *-()*), množenje (*, funkcija **()*), dijeljenje (/, funkcija */()*), inverzno dijeljenje (\, funkcija *\()*), potenciranje (^, funkcija *^()*), ostatak množenja (&, funkcija *rem()*). Za logičke tipove definiran je operator negacije koji vrijednost *true* pretvara u *false* i obrnuto. On se označava s *!*, a funkcija koju poziva je *!()*.

```
3-1     #rezultat je 2
2*3     #rezultat je 6
4/2     #rezultat je 2
2\1     #rezultat je 0.5
3^2     #rezultat je 9
7%3     #rezultat je 1
!true   #rezultat je false
```

Za svaki od numeričkih operatora u Juliji postoji i operator za ažuriranje. Operator ažuriranja mijenja varijablu s lijeve strane odgovarajućom operacijom vrijednošću s desne strane. Operator se sastoji od simbola za operator željene operacije i znaka pridruživanja.

```
julia> x=2; x+=4
6
```

Numeričke tipove je moguće uspoređivati standardnim operatorima u Juliji. Oni primaju dva argumenta i uspoređuju ih, ako je usporedba istinita daju vrijednost *true*, a ako nije *false*. Ima šest vrsta operatora za usporedbu. Oni su operatori jednakosti (==) i nejednakosti (!=), manje (<) i manje ili jednako (<=), te veće (>) i veće ili jednako (>=).

```
1==1    #rezultat je true
1!=1    #rezultat je false
1<2     #rezultat je true
1>=2    #rezultat je false
```

Julija ima dostupan veliki broj matematičkih funkcija: funkcije za zaokruživanje (*round()*, *floor()*, *ceil()*, *trunc()*,...), funkcije predznaka i apsolutnih vrijednosti (*abs()*, *sign()*, *copysign()*,...), funkcije potencija, logaritama i korijena (*sqrt()*, *cbrt()*, *exp()*, *log()*, *log2()*, *log10()*,...), trigonometrijske i hiperbolične funkcije (*sin()*, *cos()*, *tan()*, *sinh()*, *cosh()*, *tanh()*,...) i specijalne funkcije (*gamma()*, *zeta()*, *besselj()*,...).

2.5.2 Definiranje funkcija

Julija dopušta kreiranje vlastitih funkcija. Postoji nekoliko istovaljanih načina na koji se funkcija može kreirati. Funkcije se može kreirati kao jedan izraz, ali tada cijela

funkcija mora stati u jedan red.

```
f(x)=3x^2+4x+1
```

Funkciju je moguće kreirati i u bloku koji počinje s naredbom *function*.

```
function kvadrat(x)
  x*x
end
```

Kod definiranja funkcija se može koristiti naredba *return*. Ona označava koju vrijednost funkcija vraća kao rezultat. To nije uvijek potrebno, na primjer u linearno napisanim funkcijama, jer ako je ne koristimo vraća se zadnja vrijednost u funkciji. Kod funkcija koje se granaju i imaju uvjetovani rezultat mora se koristiti *return*.

```
function step(x)
  if x<1 return 0.0
  else return 1.0
  end
end
```

Funkcije u Juliji, osim obaveznih, mogu imati i opcionalne argumente. Oni se zadaju kod definiranja funkcije, a pri pozivanju se ne moraju upotrebljavati. Ako se upotrebljavaju moraju biti navedeni istim redoslijedom kojim su definirani, a ako ne funkcija će biti pozvana s zadanim vrijednostima.

```
function quad(x, a=1.0, b=1.0, c=1.0)
  a*x^2+b*x+c
end
quad(3)          #rezultat je 13
guad(2,2,2,2)    #rezultat je 14
quad(4,0,0,5)    #rezultat je 5
```

Slično se može definirati funkcija s ključnim riječima, koje su također opcionalni argument, ali ne moraju se navoditi redoslijedom kojima su definirani u funkciji jer imaju imena. Kod definiranja, funkcije se odvajaju s točkom-zarez od obaveznih argumenata.

```
function ssum(x, y; a=1.0, b=1.0)
  a*x+b*y
end
ssum(1,1)          #rezultat je 2.0
ssum(1,1,b=2)      #rezultat je 3.0
ssum(2,3,b=2,a=5) #rezultat je 16
```

2.5.3 Multiple dispatch

Multiple dispatch je jedna od glavnih značajki Julije. To je mogućnost definiranja funkcija s više metoda izvršavanja. Većina funkcija ugrađenih u Juliju imaju više metoda izvršavanja. Broj metoda koje funkcija ima možemo saznati tako da se u REPL upiše ime funkcije.

```
julia> +
+ (generic function with 171 methods)
```

Pomoću funkcije *methods()*, koja kao argument uzima ime funkcije, moguće je vidjeti sve metode od kojih se funkcija sastoji.

```
julia> methods(sin)
# 11 methods for generic function "sin":
sin(a::Complex{Float16}) at float16.jl:151
sin(z::Complex{T<:Real}) at complex.jl:548
sin(x::Float64) at math.jl:137
sin(x::Float32) at math.jl:138
sin(a::Float16) at float16.jl:150
sin(x::BigFloat) at mpfr.jl:613
sin(x::Real) at math.jl:139
sin{Tv,Ti}(A::SparseMatrixCSC{Tv,Ti}) at sparse/sparsematrix.jl:648
sin{T<:Number}}(::AbstractArray{T<:Number,1}) at operators.jl:380
sin{T<:Number}}(::AbstractArray{T<:Number,2}) at operators.jl:381
sin{T<:Number}}(::AbstractArray{T<:Number,N}) at operators.jl:383
```

Različite metode omogućavaju da funkcija radi s različitim tipovima vrijednosti. Iako je moguće konstruirati funkciju bez da se odredi tip vrijednosti koje će uzimati kao argumente, takav način konstrukcije je nepoželjan kada je cilj napisati program koji će se brzo izvršavati. Kada tipovi nisu definirani Julijin sustav dinamičkih tipova mora sam pokušati odrediti koje tipove funkcija koristi.

Ako se funkcija pozove i tipovi vrijednosti argumenata ne odgovaraju niti jednoj metodi, Julija će izbaciti grešku.

```
julia> "abc"+"def"
ERROR: MethodError: `+` has no method matching +(::ASCIIString,
::ASCIIString)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
```

Dodavanje nove metode je jednostavno. Definiramo funkciju s istim imenom kao već postojeća funkcija. U definiciji funkcije navedemo koji će se tipovi vrijednosti koristiti kao argumenti.

```
julia> function +(x::ASCIIString,y::ASCIIString)
       join([x,y])
       end
WARNING: module Main should explicitly import + from Base
+ (generic function with 172 methods)

julia> "abc"+"def"
"abcdef"
```

2.5.4 Anonimne funkcije

Anonimne funkcije su funkcije bez imena. Definiraju se simbolom ->.

```
julia> x->x^2
(anonymous function)
```

Najkorisnije su kada želimo proslijediti funkciju kao argument drugoj funkciji. Funkcija *map()* kao rezultat vraća niz na koji su mapirani rezultati izvršavanja funkcije za pojedine vrijednosti u nizu koji je bio ulazni argument.

```
julia> map(x->x+1,[0,1,2])
3-element Array{Int32,1}:
 1
 2
 3
```

2.6 *Kontrola toka*

Kontrola toka u programiranju se svodi na uvjetno izvršavanje i ponavljano izvršavanje. Uvjetno izvršavanje je u Juliji izvedeno *if* izrazom. Ponavljano izvršavanje je riješeno petljama. Julia ima dvije vrste petlji: *for* i *while*. Za razliku od nekih drugih programskih jezika, Julija nema repeat petlju, ali se njezino ponašanje može simulirati *while* petljom.

2.6.1 Uvjetno izvršavanje

Blok *if* u Juliji je dio sintakse izraza *if-elseif-else*. Koristi se za uvjetovano izvršavanje. Ako je uvjet zadovoljen (daje vrijednost *true*) izvršava se izraz ili blok izraza. Blok izraza se zatvara ključnom riječju *end*.

```
if a==b println("a i b su jednaki") end
```

```
if a!=b
    println("a i b vise nisu razliciti")
    a=b
end
```

Ostali izrazi *if-elseif-else* sintakse su blokovi *elseif* i *else*. Uvjet *elseif* bloka se provjerava, ako uvjet *if* bloka nije zadovoljen. Ako je uvjet *elseif* bloka zadovoljen on se izvršava. Sintaksa ne ograničava koliko *elseif* blokova može biti. Ako nijedan uvjet nije zadovoljen izvršava se *else* blok.

```
if a<1
    println("a je manje od 1")
elseif a>1
    println("a je vece od 1")
else
    println("a je 1")
end
```

2.6.2 For petlja

For petlja se definira naredbom *for*. Ona iterira po vrijednostima koje mogu biti bilo kojeg formata koji sadrže niz vrijednosti. Najčešće su to rasponi i nizovi. Nakon što je naveden objekt po kojem se iterira, navodi se blok izraza koje petlja izvršava, a koji se završava naredbom *end*. Varijabla kojom iteriramo je vidljiva samo unutar petlje, izvan nje ne postoji.

```
for i=1:5
    print("$i ")
end

for i in ["a","12","crveno"]
    println(i)
end
```

Korak u izvrašavanju petlje se može preskočiti pomoću naredbe *continue*.

```
for i=1:10
    if rem(i,2)!=0 continue end
    print("$i ")
end
```

2.6.3 While petlja

While petlja je tip petlje koji se izvršava sve dok je proizvoljni uvjet zadovoljen. Petlja se definira naredbom *while* nakon koje slijedi uvjet. Nakon toga je blok naredbi koje se izvršavaju sve dok je uvjet zadovoljen. Petlja se zatvara naredbom *end*. Unutar petlje se obično nalazi izraz koji će utjecati na istinitost uvjeta. Inače je moguće konstruirati beskonačnu petlju.

```
x=0
while x<=2
    print("$x ")
    x+=1
end
```

Neki programski jezici su imali repeat petlju. Takva petlja u Juliji ne postoji, ali je moguće konstruirati petlju koja se ponaša na sličan način. Repeat petlja se izvršava sve dok neki uvjet nije zadovoljen, nakon čega se prekida. Julija ima naredbu *break* koja završava izvođenje petlje. Ako se while petlji kao uvjet postavi vrijednost *true* ona će se uvijek izvršavati, da bi se to izbjeglo koristi se izraz *break* koji se izvrši kad je neki uvjet zadovoljen.

```
x=0
while true
    print ("$x ")
    x+=1
    if x>2 break end
end
```

2.7 I/O

Julijin sustav za ulaz/izlaz (input/output) je orijentiran na tokove (stream). Julija čita ili piše tokove bajtova. Standardne varijable za ulaz i izlaz, defnirane u REPL-u, su *STDIN*, *STDOUT* i *STDERR*. To su tokovi tipa *Base.TTY*, gdje je TTY kratica od teletype. Za jednostavan upis ili ispis se koriste funkcije *read()* i *write()*. Funkcija *write* nakon ispisa niza znakova, vraća i broj znakova koje je ispisala, uključujući posebne znakove kao što su $\r$(carriage return - vraća kursor na početak reda) i $\n$(line feed - prelazi u novi red).

```
read(STDIN, Char)           #ceka upis jednog znaka
read(STDIN, Int32)          #nakon upisa 123 vraca 221458993
write(STDOUT, "ispis\r\n")  #ispisuje niz znakova i 7
```

Za upis proizvoljnog tipa vrijednosti treba koristiti funkciju *readline()*. Ona kao vrijednost upisa vraća niz znakova sa specijalnim znakovima, ali je njih moguće uk-

loniti funkcijom *chomp()*. Ostatak se funkcijom *parse()* može promijeniti u željeni tip.

```
function get(t::Type)
    in=chomp(readline(STDIN))
    return parse(t,in)
end
a=get(Int32)    #za unos 123 daje 123 tipa Int32
```

Za čitanje i pisanje u datoteke na disku potrebno je prvo otvoriti datoteku. Za to služi funkcija *open()*. Ona kao argument prima ime datoteke na disku, na primjer ako je datoteka u Julijinoj radnoj mapi onda se navodi samo ime (”primjer.txt”), a ako je bilo gdje na disku treba unijeti punu putanju sa simbolima koji označavaju posebne znakove (”C:\\primjer.txt”). Kao opcionalni argument možemo navesti način otvaranja *”r”* za čitanje, *”w”* za pisanje (ako već postoji datoteka će biti prepisana) i *”a”* za pisanje na kraj datoteke. Nakon što se završi rad s datotekom potrebno ju je zatvoriti funkcijom *close()*.

```
io1=open("primjer.txt","r")
close(io1)
```

Cijelu datoteku je moguće pročitati funkcijom *readlines()*. Svaki red datoteke će biti pohranjen u niz tipa vrijednosti *ByteString* i sadržavati će specijalne znakove. Specijalne znakove je moguće ukloniti funkcijom *chomp()*.

```
io1=open("primjer.txt","r")
data=readlines(io1)
for i=1:length(data)
    line=chomp(data[i])
    println("$i. red: $line")
end
close(io1)
```

Osim čitanja u datoteke je moguće i pisati. Postupak je sličan kao i za čitanje. Prvo se otvara datoteka, zatim u nju upisujemo nizove znakova i na kraju zatvaramo datoteku. Funkcija *println()* je u stanju ispisivati u datoteke.

```
io2=open("primjer2.txt","w")
for i=1:5
    println(io2,"$i. red")
end
close(io2)
```

2.8 Pozivanje vanjskih funkcija

Pozivanje funkcija iz Pythona se jednostavno izvodi pomoću paketa *PyCall*. Paket pozivamo naredbom *using*. Python se onda može koristiti iz REPL-a ili skripte. Moguće je koristiti izvršavanje iz Pythona pomoću funkcije *pyeval()*, a i uvesti Pythonove biblioteke pomoću makro-naredbe *@pyimport*. Nakon što se uvezu, Pythonove biblioteke se pozivaju na isti način kao i u Pythonu.

```
using PyCall
pyeval("3+2")    #rezultat je 5
@pyimport math
math.cos(pi)     #rezultat je -1.0
```

3 Vizualizacija podataka i obrada rezultat mjerenja

3.1 Paket DataFrames

DataFrames je paket koji proširuje Julijinu funkcionalnost rada s tabličnim podacima. Ovaj paket posjeduje tri bitna elementa: vrijednost *NA*, *DataArray* proširenje tipa *Array* i *DataFrame* tip koji je pogodan za spremanje podataka u tabličnom obliku. *DataFrames* je pogodan za učitavanje i ispis podataka u datoteke, te olakšava rad s nekim paketima za grafiku kao što je *Gadfly*. Paket se instalira iz REPL-a naredbom *Pkg.add("DataFrames")*, a možemo ga koristiti nakon naredbe *using DataFrames*.

Vrijednost *NA* (engl. not available = nije dostupno) označava vrijednost koja nedostaje. Bilo koja operacija u kojoj se koristi *NA* vraća isto vrijednost *NA*. Ta vrijednost označava neznanje o podatku, tako da se iz niti jedne operacije s *NA* ne može nešto saznati.

```
using DataFrames
NA+1    # rezultat je NA
```

Vrijednost *NA* nije moguće spremiti u Juliji u običan niz. Zato ovaj paket uvodi niz tipa *DataArray*. Njega se može konstruirati makro-naredbom *@data()*. Takav niz može sadržavati vrijednost *NA*. S obzirom da operacije s *NA* uvijek vraćaju *NA*, na nizu koji ga sadrži moguće je upotrijebiti funkciju *dropna()*, koja uklanja sve vrijednosti *NA*.

```
da=@data([1,2,NA,4])
sum(da)           # rezultat je NA
sum(dropna(da))   # rezultat je 7
```

Za skupove podataka, koji nemaju jednostavnu strukturu koja se može pohraniti u niz, postoji struktura tipa *DataFrame*. Ona se sastoji od stupaca i redova. Redovi su indeksirani od broja 1, a stupcima je moguće i zadati imena. Setovi podataka se mogu sastojati od stupaca različitih tipova vrijednosti, ali jedan stupac u sebi sadrži isti tip vrijednosti. Svi stupci u skupu podataka su iste dužine, te ih je moguće pozivati ili imenom ili indeksom. Skup podataka se gradi funkcijom *DataFrame()*

```
julia> df=DataFrame(A=[1.0,2.0,3.0],B=["jedan","dva","tri"],C=[true,
true,true])
3x3 DataFrames.DataFrame
| Row | A   | B       | C    |
|-----|-----|---------|------|
| 1   | 1.0 | "jedan" | true |
| 2   | 2.0 | "dva"   | true |
| 3   | 3.0 | "tri"   | true |
```

Moguće je i pristupiti samo određenim stupcima (indeksom ili imenom).

```
julia> df[1]
3-element DataArrays.DataArray{Float64,1}:
 1.0
 2.0
 3.0

julia> df[:C]
3-element DataArrays.DataArray{Bool,1}:
 true
 true
 true
```

Za pristup redovima koriste se indeksi redova. Oni se navode prije indeksa stupca. Moguće je koristiti i intervale (naravno, to je bilo moguće i za stupce).

```
julia> df[2:3,:]
2x3 DataFrames.DataFrame
| Row | A   | B       | C    |
|-----|-----|---------|------|
| 1   | 2.0 | "dva"   | true |
| 2   | 3.0 | "tri"   | true |
```

Za prikaz samo prvih ili zadnjih šest redova, koriste se funkcije *head()*, odnosno *tail*. Funkcijom *describe()* moguće je dobiti opis podataka.

Paket *DataFrames* je pogodan i za rad s datotekama. Funkcije *writetable()* i *readtable* ispisuju, odnosno unose podatke iz posebno formatiranih datoteka (CSV, engl. comma separated values = zarezom odvojene vrijednosti). Za ispis u datoteku je dovoljno funkciji kao argument navesti samo ime datoteke i podatke koje želimo ispisati. Ali postoje i opcionalni argumenti, kao što su znak koji želimo koristiti za odvajanje vrijednosti i želimo li ispisati zaglavlje.

```
julia> writetable("data.csv",df,separator=',',header=false)
```

Za unos podataka iz datoteke, funkcija *readtable()* koristi slične argumente. Ali, potrebno je znati kako je datoteka formatirana, ima li zaglavlje i koji je znak korišten za odvajanje vrijednosti.

```
julia> df=readtable("data.txt",separator='\t',header=false);
```

3.2 Paketi za grafiku

Julija u svojoj osnovi nema podršku za stvaranje grafike, nego je ona izvedena pomoću mnogobrojnih paketa, kao što su *Winston*, *Gadfly*, *PyPlot*, *ASCIIPlots* i drugi. Najzanimljiviji su *Gadfly*, koji ima izravnu podršku za paket *DataFrames*, i *PyPlot*, koji se oslanja na Python i njegovu izvrsnu biblioteku matplotlib.

3.2.1 Gadfly

Gadfly se instalira naredbom *Pkg.add("Gadfly")*. Podržava brojne oblike plotova, koji se mogu prikazati u zadanom web pregledniku. Za ispis u datoteku koristi mnoge formate (SVG, PNG, PDF, PS, PGF). Glavni nedostaci ovog paketa su što nema podršku za LaTeX nizove znakova i zasad još nema podršku za trodimenzionalne plotove, dok su glavne prednosti rad s paketom *DataFrames* i čist, moderan izgled plotova. Koristiti ga možemo nakon naredbe *using Gadfly*.

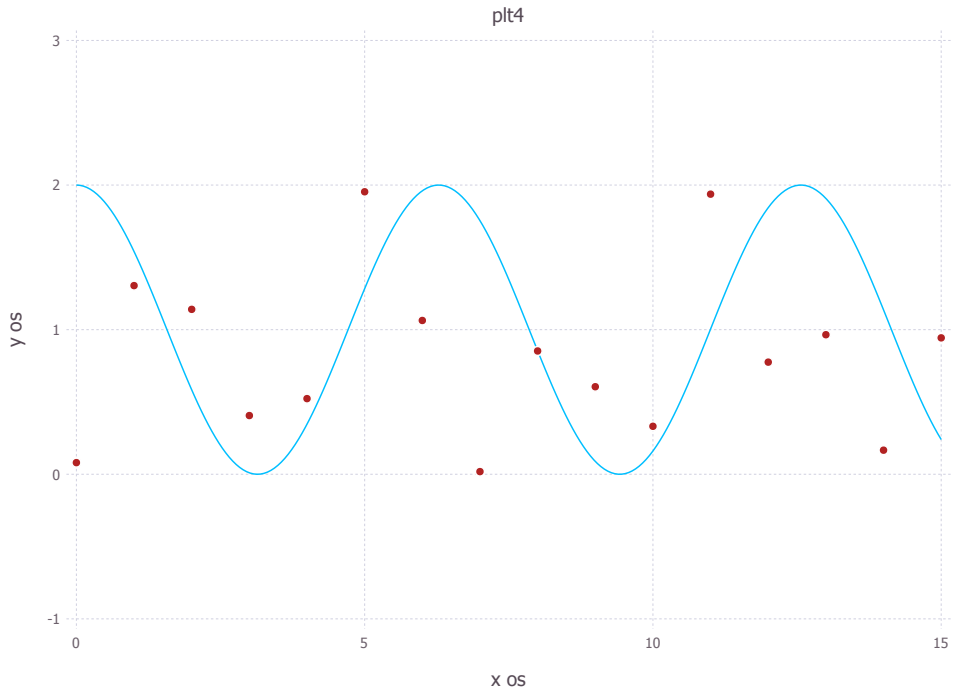
Pomoću funkcije *plot()* mogu se nacrtati grafovi funkcija, raspršenja točaka, putanje i slični dvodimenzionalni grafovi. Podržan je veći broj slojeva (layer) tako da je moguće i više grafova spojiti u jedan.

```
using Gadfly
a=[1.0,2.0,3.0,4.0,5.0];
b=[2.0,3.0,-1.0,2.0,0.0];
plt1=plot(sin,0,15)
plt2=plot(x=a,y=b,Geom.point)
plt3=plot(x=a,y=b,Geom.line)
plt4=plot(layer(x=0:15,y=2*rand(16),Geom.point,
                Theme(default_color=colorant"firebrick")),
          layer(x->1+cos(x),0,15,
                Theme(default_color=colorant"deepskyblue")),
          Guide.xlabel("x os"),Guide.ylabel("y os"),
          Guide.title("plt4"),
          Coord.cartesian(xmin=0.0,xmax=15.0,ymin=-1.0,ymax=3.0));
```

Argumenti u funkciji *plot()* koji počinju s *Geom.* određuju tip grafa koji će biti nacrtan. Neki od njih su:*Geom.point* (crta točke), *Geom.line* (povezuje točke linijama), *Geom.path* (crta putanje), a *Geom.contour* (crta ekvipotencijalne linije). Argumenti koji počinju s *Guide.* označavaju anotaciju na grafu. Za određivanje granica grafa koristi se objekt *Coord.cartesian*, u kojem se osim granica može odrediti i omjer slike varijablom *aspect_ratio*. Također je moguće mijenjati temu (*Theme*) svakog sloja čime mu mijenjamo izgled.

Za ispis u datoteku se koristi funkcija *draw()*. Ona kao argument prima: format ispisa (koji je također funkcija u kojoj navodimo ime datoteke i veličinu ispisa) i plot koji želimo ispisati.

```
draw(PDF("gadflyplot.pdf",20cm,15cm),plt4)
```



Slika 3.1: Primjer grafa nacrtanog s paketom Gadfly

3.2.2 PyPlot

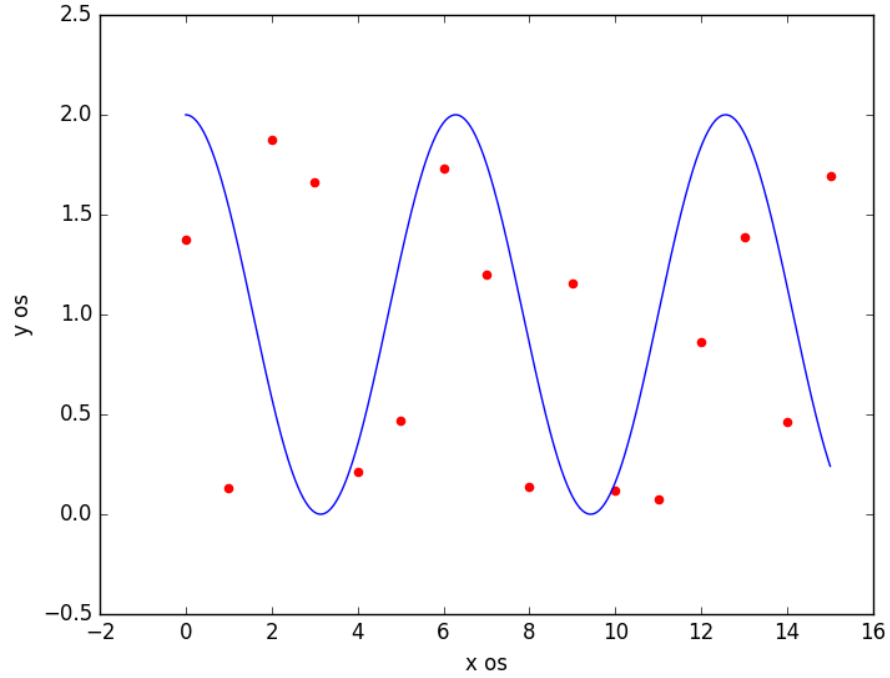
PyPlot je paket za grafiku temeljen na Pythonovoj biblioteci matplotlib i stoga zahtjeva instalaciju Pythona i navedene biblioteke. Za razliku od paketa *Gadfly*, zna raditi s 3D plotovima i podržava LaTeX nizove znakova, ali je vizualno neugledniji. Instalira se naredbom *Pkg.add("PyPlot")*, a možemo ga koristiti nakon naredbe *using PyPlot*.

Osnovna funkcija koju koristimo za crtanje grafova je *plot()*, a za crtanje raspršenja točaka *scatter()*. Ona kao argumente prima nizove brojeva iz kojih konstruira graf. Opcionalni argumenti su boja, debljina i stil linije. Funkcije *title()*, *xlabel()* i *ylabel()* imenuju naslov i osi grafa. Pomoću funkcije *legend()* se mogu anotirati pojedine komponente grafa.

```
using PyPlot
x1=0:1:15; y1=2*rand(16);
x2=linspace(0,15,1000); y2=1+cos(x2);
scatter(x1, y1, color="red")
plot(x2, y2, color="blue")
xlabel("x os")
```

```
ylabel("y os")
savefig("pyplot.png")
```

Funkcija *savefig()* sprema sliku u SVG ili PNG datoteku. Paket *PyPlot* još sadrži



Slika 3.2: Primjer grafa nacrtanog s paketom PyPlot

funkcije za crtanje grafova *contour()*, *plot3D()*, *scatter3D()*, *contour3D()*, *plot_surface()* i druge.

3.3 Obrada rezultata mjerenja

3.3.1 Mjerenje jedne veličine

Neko mjerenje se obavi n puta. Time se dobiva uzorak od n elemenata od populacije. Najvjerojatnija vrijednost mjerene veličine bit će aritmetička sredina svih mjerenja. Standardna devijacija uzorka je veličina koja opisuje preciznost mjerenja. Standardna devijacija aritmetičke sredine se još naziva nepouzdanost mjerenja ili

standardna pogreška. Pomoću nje je jednostavno definirati i relativnu pogrešku.

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{3.1}$$

$$m = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} \tag{3.2}$$

$$M = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n(n-1)}} = \frac{m}{\sqrt{n}} \tag{3.3}$$

$$R = \frac{M}{\overline{x}} \cdot 100\% \tag{3.4}$$

S obzirom na to da Julija sadrži funkcije *mean()* i *std()*, gdje je *mean* aritmetička sredina, a *std* standardna devijacija uzorka, sve četiri veličine se jednostavno mogu definirati u jednoj funkciji.

```
function mjerenje(x)
    # najvjerojatnija vrijednost mjerene velicine
    xs=mean(x)
    # preciznost mjerenja
    m=std(x)
    # nepouzdanost mjerenja
    M=m/sqrt(length(x))
    # relativna pogreska
    R=M/xs*100
    return [xs,m,M,R]
end
```

Kada se na neko mjerenje, na primjer duljine, primjeni definirana funkcija *mjerenje()*, ona vraća sve četiri veličine.

```
julia> l=[8.0, 8.1, 8.0, 8.1, 7.9, 8.0, 8.0, 8.2]; mjerenje(l)
4-element Array{Float64,1}:
 8.0375
 0.0916125
 0.0323899
 0.402985
```

3.3.2 Metoda najmanjih kvadrata

Za neka mjerenja veličina koje nisu nezavisne, i iako iz teorije predviđamo linearnu ovisnost, moguće je provesti metodu najmanjih kvadrata.

```
l=[0.1, 0.2, 0.3, 0.4, 0.5]
r1=[340.0, 650.0, 980.0, 1340.0, 1640.0]
```

Metoda najmanjih kvadrata daje, na osnovu mjerenja, smjer pravca i odsječak na osi y. Za pravac $y = ax + b$:

$$a = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2} \tag{3.5}$$

$$b = \frac{\sum x_i^2 \sum y_i - \sum x_i \sum x_i y_i}{n\sum x_i^2 - (\sum x_i)^2} \tag{3.6}$$

$$M_a = \sqrt{\frac{1}{n-2}\left(\frac{n\sum y_i^2 - (\sum y_i)^2}{n\sum x_i^2 - (\sum x_i)^2} - a^2\right)} \tag{3.7}$$

$$M_b = M_a\sqrt{\frac{\sum x_i^2}{n}} \tag{3.8}$$

Prvo *for* petlja računa sve izraze za sume, a nakon toga se računaju koeficijent smjera i odsječak, te njihove pogreške. Rezultati se na kraju spremaju u matricu.

```
N=length(l)
X=Y=XX=YY=XY=0.0;
for i=1:N
    X+=l[i]
    Y+=r1[i]
    XX+=l[i]*l[i]
    YY+=r1[i]*r1[i]
    XY+=l[i]*r1[i]
end

a=(N*XY-X*Y)/(N*XX-X*X);
Ma=sqrt((1/(N-2))*((N*YY-Y*Y)/(N*XX-X*X)-a*a));
b=(XX*Y-X*XY)/(N*XX-X*X);
Mb=Ma*sqrt(XX/N);
koef=[a Ma;b Mb]
```

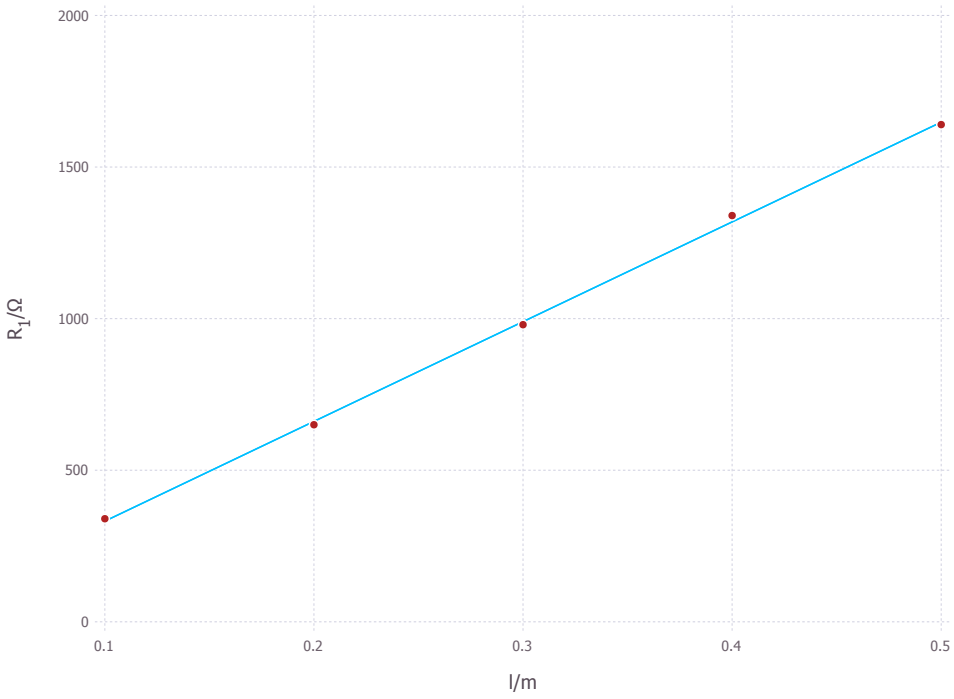
Pomoću koeficijenta nagiba i odsječka na y osi konstruiramo funkciju pravca koji je dobiven linearnom regresijom.

```
function pravac(x; a=koef[1], b=koef[2])
    return a*x+b
end
```

Na kraju pomoću paketa *Gadfly* nacrtamo graf na kojem se nalaze točke koje predstavljaju mjerenja i pravac čiji su smjer i odsječak na y osi dobiveni metodom najmanjih kvadrata.

```
using Gadfly
plt=plot(layer(x=l,y=r1,Geom.point,
        Theme(default_color=colorant"firebrick")),
```

```
    layer(pravac,0.1,0.5,Geom.line,
          Theme(default_color=colorant"deepskyblue")),
    Coord.Cartesian(xmin=0.1, xmax=0.5),
    Guide.XLabel("l/m"),
    Guide.YLabel("R<sub>1</sub>/\u03a9"));
draw(PDF("mjerenja2.pdf", 20cm, 15cm),plt)
```



Slika 3.3: Pravac dobiven metodom najmanjih kvadrata za ovisnost otpora o duljine žice

4 Problem gušenog harmoničkog oscilatora

4.1 Opis problema

Harmonički oscilator je uređaj koji se sastoji od mase m i opruge konstante k. Kada se masa izmakne iz ravnotežnog položaja opruga ju nastoji vratiti, što dovodi do titranja mase.

$$m\ddot{x} = -kx \tag{4.1}$$

U slučaju da postoji otpor proporcionalan s brzinom, jednadžba dobiva još jedan član.

$$m\ddot{x} = -kx - \eta\dot{x} \tag{4.2}$$

Radi pojednostavljenje, uvodimo konstante a i b.

$$a = \frac{k}{m} \tag{4.3}$$

$$b = \frac{\eta}{m} \tag{4.4}$$

$$\ddot{x} = -ax - b\dot{x} \tag{4.5}$$

Jednadžba harmoničkog oscilatora je obična diferencijalna jednadžba drugog reda.

4.2 Runge-Kutta metoda

Runge-Kutta metode se temelje na Taylorovom razvoju i daju općenito dobre algoritme za rješavanje običnih diferencijalnih jednadžbi. Jednadžba

$$\frac{dy}{dt} = f(y,t), \tag{4.6}$$

se riješava u koracima i s

$$y(t) = \int f(y,t)dt, \tag{4.7}$$

imamo

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(y,t)dt. \tag{4.8}$$

Osnovno načelo je izračunavanje međukoraka za y_{i+1}. Zbog računanja tog međukoraka više je operacija koje treba obaviti, ali i veća stabilnost rješenja koje se dobije. Algoritam Runge-Kutta metode četvrtog reda izračunava četiri koeficijenta koji se onda po sljedećoj formuli zbrajaju:

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{4.9}$$

gdje su za veličinu koraka h koeficijenti:

$$k_1 = hf(y_i, t_i), \tag{4.10}$$

$$k_2 = hf(y_i + k_1/2, t_i + h/2), \tag{4.11}$$

$$k_3 = hf(y_i + k_2/2, t_i + h/2), \tag{4.12}$$

$$k_4 = hf(y_i + k_3, t_i + h). \tag{4.13}$$

4.3 Numeričko rješenje

Runge-Kutta metoda je namijenjena rješavanju sustava običnih diferencijalnih jednadžbi prvog reda. U slučaju harmoničkog oscilatora imamo diferencijalnu jednadžbu drugog reda. Uvođenjem jednostavne supstitucije, možemo je svesti na sustav dvije diferencijalne jednadžbe prvog reda. Tada možemo primjeniti Runge-Kutta metodu za rješavanje problema harmoničkog oscilatora.

Supstitucija je definicija brzine $\dot{x} = v$ čime se dobivaju dvije jednadžbe koje treba numerički riješiti.

$$\dot{x} = v \tag{4.14}$$

$$\dot{v} = -ax - bv \tag{4.15}$$

Za te dvije jednadžbe u Juliji se definiraju odgovarajuće funkcije.

```
function dx(v::Float64)
    return v
end

function dv(x::Float64, v::Float64; a=4.0::Float64, b=0.5::Float64)
    return (-a*x-b*v)
end
```

Kako imamo dvije diferencijalne jednadžbe prvog reda za rješenje svake treba po jedan početni uvjet. U ovom slučaju su to položaj i brzina. Osim početnih uvjeta potrebno je definirati i interval vremena za koji tražimo rješenje, te korak u Runge-Kutta metodi.

```
x,v=[1.0,1.0]      #pocetni uvjeti (polozaj i brzina)
ti=0.0             #pocetni trenutak
tf=15.0            #zavrsni trenutak
h=(tf-ti)/100000   #korak
```

Za spremanje podataka je pogodna varijabla tipa *DataFrame*. U nju će se spremati po tri varijable za svaki korak izračuna: vrijeme, položaj i brzina.

```
osc=DataFrame(vrijeme=Float64[ti],polozaj=Float64[x],brzina=Float64[v])
```

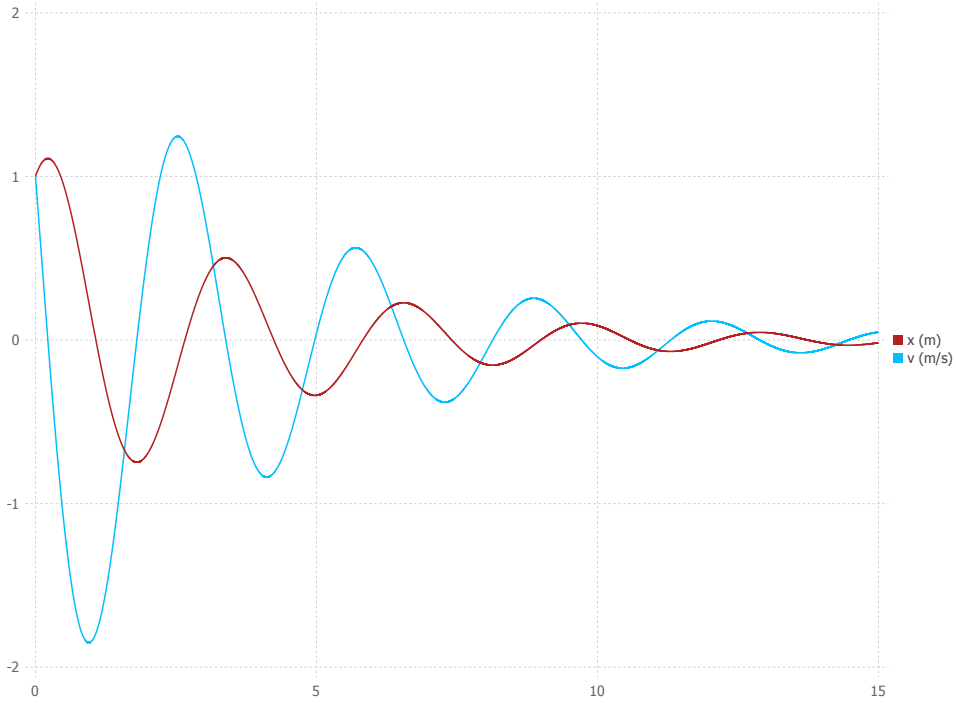
Runge-Kutta metodu provodimo pomoću *for* petlje, koja nakon izračuna za svaki definirani trenutak dodaje novi rezultat u varijablu *osc*. Prvi korak petlje se preskače jer su početni uvjeti već definirani prilikom stvaranja varijable *osc*.

```
for t=ti:h:tf
    if t==ti
        continue
    else
```

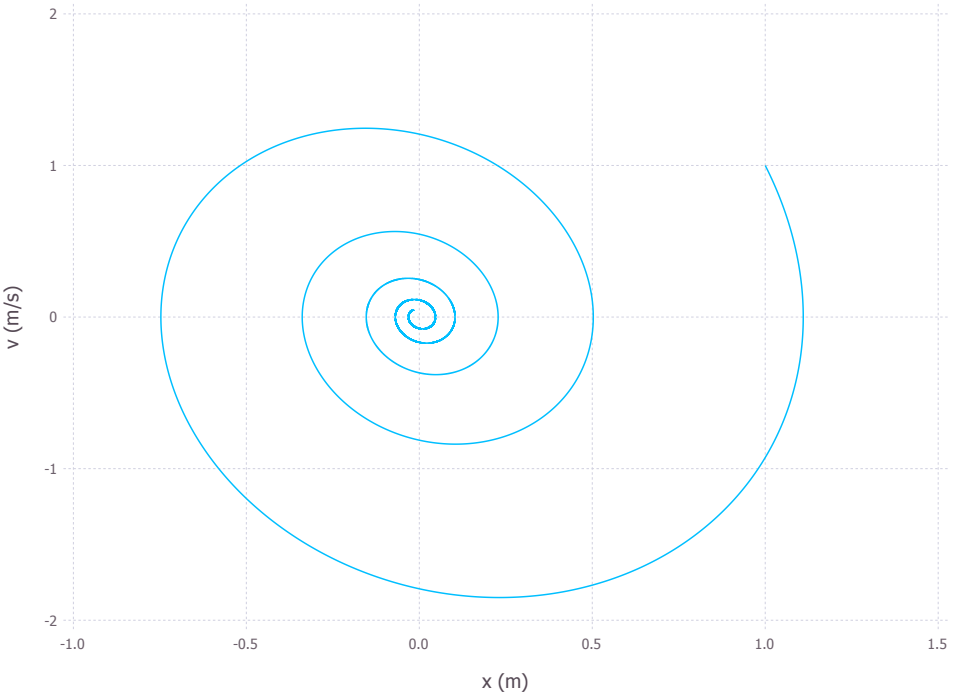
```
        k1x=h*dx(v)
        k2x=h*dx(v)
        k3x=h*dx(v)
        k4x=h*dx(v)
        x+=(1.0/6.0)*(k1x+2*k2x+2*k3x+k4x)
        k1v=h*dv(x, v)
        k2v=h*dv(x, v+k1v/2)
        k3v=h*dv(x, v+k2v/2)
        k4v=h*dv(x, v+k3v)
        v+=(1.0/6.0)*(k1v+2*k2v+2*k3v+k4v)
    end
    push!(osc[:vrijeme],t)
    push!(osc[:polozaj],x)
    push!(osc[:brzina],v)
end
```

Rezultat je skup podataka u kojem su definirani položaj i brzina. Taj rezultat se može grafički prikazati u ovisnosti o vremenu i u faznom prostoru pomoću paketa *Gadfly*.

```
plt1=plot(layer(osc,x="vrijeme",y="polozaj",Geom.line,
          Theme(default_color=colorant"firebrick")),
          layer(osc,x="vrijeme",y="brzina",Geom.line,
          Theme(default_color=colorant"deepskyblue")),
    Coord.Cartesian(xmin=ti, xmax=tf),
    Guide.XLabel(nothing),
    Guide.YLabel(nothing),
    Guide.manual_color_key(" ",["x (m)","v (m/s)"],
                           ["firebrick","deepskyblue"]));
draw(PDF("osct.pdf", 20cm, 15cm),plt1)

plt2=plot(osc,x="polozaj",y="brzina",Geom.path,
     Guide.XLabel("x (m)"),
     Guide.YLabel("v (m/s)"));
draw(PDF("oscf.pdf",20cm,15cm),plt2)
```



Slika 4.1: Rješenje za položaj i brzinu harmoničkog oscilatora



Slika 4.2: Rješenje harmoničkog oscilatora u faznom prostoru

5 Problem čestice u potencijalu

5.1 Opis problema

Gibanje čestice u nekom potencijalu opisano je diferencijalnim jednadžbama. Ako na česticu mase m u ravnini djeluje potencija $U(x, y)$, moguće je napisati jednadžbe gibanja jer negativni gradijent potencijala daje silu.

$$F_x = -\frac{\partial}{\partial x}U(x, y) \tag{5.1}$$

$$F_y = -\frac{\partial}{\partial y}U(x, y) \tag{5.2}$$

Iz toga se dobivaju dvije diferencijalne jednadžbe drugog reda.

$$m\ddot{x} = -\frac{\partial}{\partial x}U(x, y) \tag{5.3}$$

$$m\ddot{y} = -\frac{\partial}{\partial y}U(x, y) \tag{5.4}$$

Za rješavanje takvih jednadžbi u Juliji može se koristiti *ODE* paket.

5.2 ODE paket

Paket *ODE* služi za rješavanje diferencijalnih jednadžbi. Cijeli paket je pisan isključivo u Juliji i može rješavati linearne i nelinearne diferencijalne jednadžbe. Paket se sastoji od nekoliko prilagođavajućih funkcija za rješavanje diferencijalnih jednadžbi. Da bi se greška što više smanjila, algoritam svake od funkcija kontrolira veličinu koraka pri izračunu (za razliku od Runge-Kutta metode koja koristi korak stalne veličine).

Funkcije koji čine paket su: *ode23()*, *ode45()* i *ode78()*. Prva znamenka označava red funkcije za rješavanje, a druga red kontrole pogreške. Prva koristi Bogacki-Shampine koeficijente, druga Dromand-Prince koeficijente, a treća Fehlberg koeficijente.

Sve funkcije kao argumente primaju funkciju koja sadrži niz diferencijalnih jednadžbi oblika $dy/dt = F(t, y)$, niz početnih uvjeta (za svaku od jednadžbi po jedan), te vremenski interval u kojem želimo rješenje.

```
t,y=ode45(F,y0,tspan)
```

Kao rješenje dobivamo dva niza. Prvi niz čine vrijednosti vremena, a drugi niz je višedimenzionalan i sadrži rješenje za svaki trenutak vremena prvog niza za svaku diferencijalnu jednadžbu zadanu funkcijom F.

5.3 Numeričko rješenje

Na česticu u ravnini djeluje potencijal $U(x, y)$.

$$U(x, y) = -4x^2y^3 \tag{5.5}$$

Za česticu mase $m = 1$ gibanje u ravnini je opisano dvjema diferencijalnim jednadžbama.

$$\ddot{x} = 8xy^3 \tag{5.6}$$
$$\ddot{y} = 12x^2y^2 \tag{5.7}$$

S obzirom da funkcije iz paketa *ODE* rješavaju skup diferencijalnih jednadžbi prvog reda, uvrštavajući definicije brzina $v_x = \dot{x}$ i $v_y = \dot{y}$, dobiva se set od četiri diferencijalne jednadžbe prvog reda koje opisuju gibanje čestice u ravnini.

$$\dot{x} = v_x \tag{5.8}$$
$$\dot{v}_x = 8xy^3 \tag{5.9}$$
$$\dot{y} = v_y \tag{5.10}$$
$$\dot{v}_y = 12x^2y^2 \tag{5.11}$$

U Juliji se definira funkcija koja će vraćati niz diferencijalnih jednadžbi.

```
function newton(t, n)
    (x,vx,y,vy)=n
    d_x=vx
    d_vx=8*x*y^3
    d_y=vy
    d_vy=12*x^2*y^2
    [d_x,d_vx,d_y,d_vy]
end
```

Dalje se definiraju početni uvjeti. Za rješavanje četiri diferencijalne jednadžbe prvog reda potrebna su četiri početna uvjeta: početni položaj u x smjeru, početna brzina u x smjeru, početni položaj u y smjeru i početna brzina u y smjeru. Također se definira i vremenski interval u kojem se traži rješenje.

```
start=[-1.0,2.0,-1.0,-3.0]
time=0.0:0.001:1.43
```

Sada su definirani svi argumenti koje traži funkcija *ode45()*. Nakon što funkcija riješi sustav jednadžbi, rješenja se mogu mapirati na varijable s kojima će se dalje raditi, pomoću funkcije *map()*.

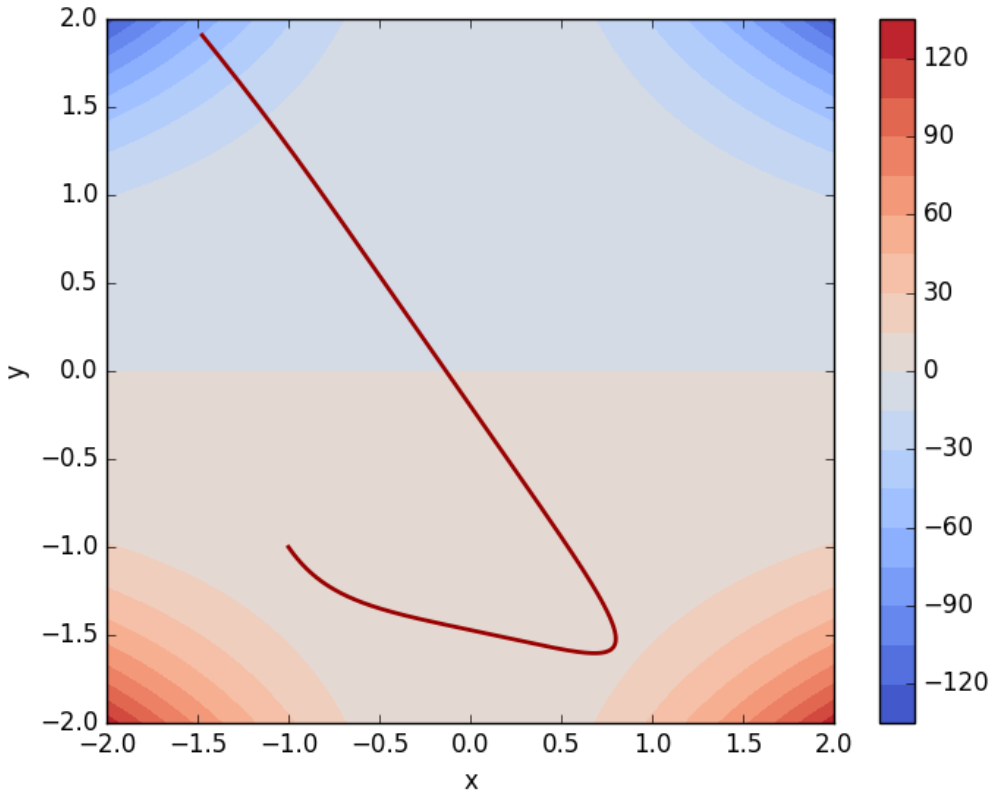
```
using ODE
t,n=ode45(newton,start,time)
xtraj=map(n->n[1],n)
ytraj=map(n->n[3],n)
```

Paket *PyPlot* pomoću funkcija *contourf()* i *plot()* može vizualizirati gibanje čestice u potencijalu. Funkcija *contourf()* je korisna za prikaz potencijala. Ona crta ekvipotencijale linije i popunjava prostor između njih odabranom raspodjelom boja. S

obzirom da funkcija *contourf()* kao argumente prima jednodimenzionalne nizove za osi i dvodimenzionalni niz za vrijednosti potrebno ih je definirati. Varijable *xpot* i *ypot* će predstavljati osi, a *zpot* vrijednosti potencijala.

```
N=300
xpot=linspace(-2,2,N)
ypot=linspace(-2,2,N)
zpot=rand(N,N)

for i=1:N
    for j=1:N
        zpot[j,i]=-4*xpot[i]^2*ypot[j]^3
    end
end
```

Funkcija *plot()* crta putanju čestice iz rješenja dobivenih funkcijom *ode45()*.



Slika 5.1: Rješenje za gibanje čestice u potencijalu

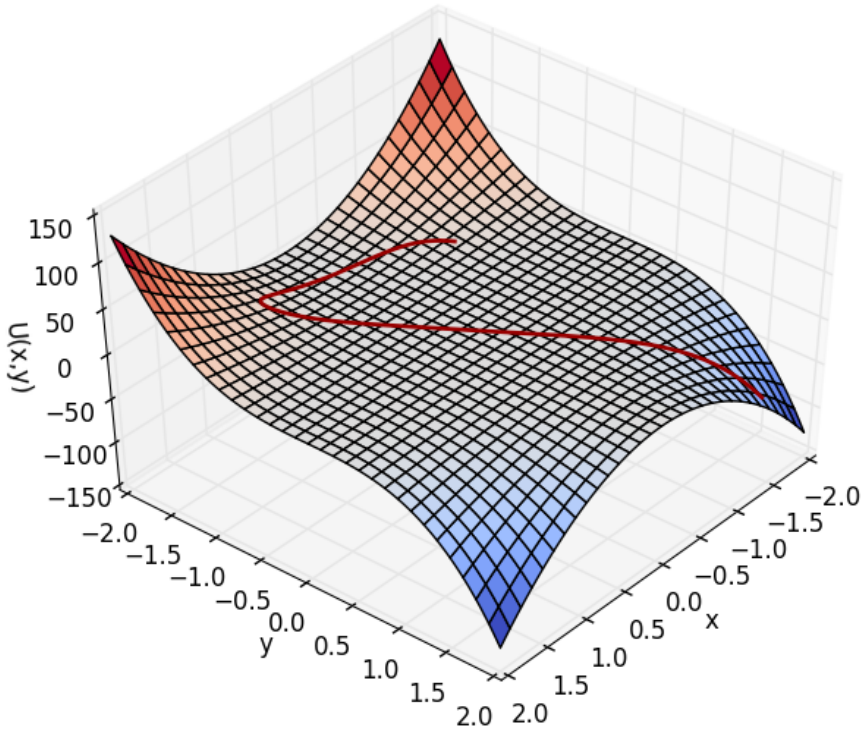
```
using PyPlot
figure()
contourf(xpot,ypot,zpot,20,cmap=ColorMap("coolwarm"))
```

```
colorbar()
xlabel("x")
ylabel("y")
plot(xtraj,ytraj,color="#990000",linewidth=2)
savefig("cestupot2d.png")
```

Pomoću funkcije *savefig()* numeričko rješenje se sprema u datoteku.

S obzirom da paket *PyPlot* može crtati i trodimenzionalne grafove, moguće je predstaviti potencijal kao površinu na kojoj se čestica giba. Za to se može koristiti funkcija *plot_surface()* za potencijalnu plohu i funkcija *plot3D* za prikaz putanje čestice na plohi. Za prikaz putanje čestice na plohi potrebno je primijeniti dobivena rješenja na plohu.

```
zsurf=-4.*xtraj.^2.*ytraj.^3
```



Slika 5.2: Rješenje za gibanje čestice na potencijalnoj plohi

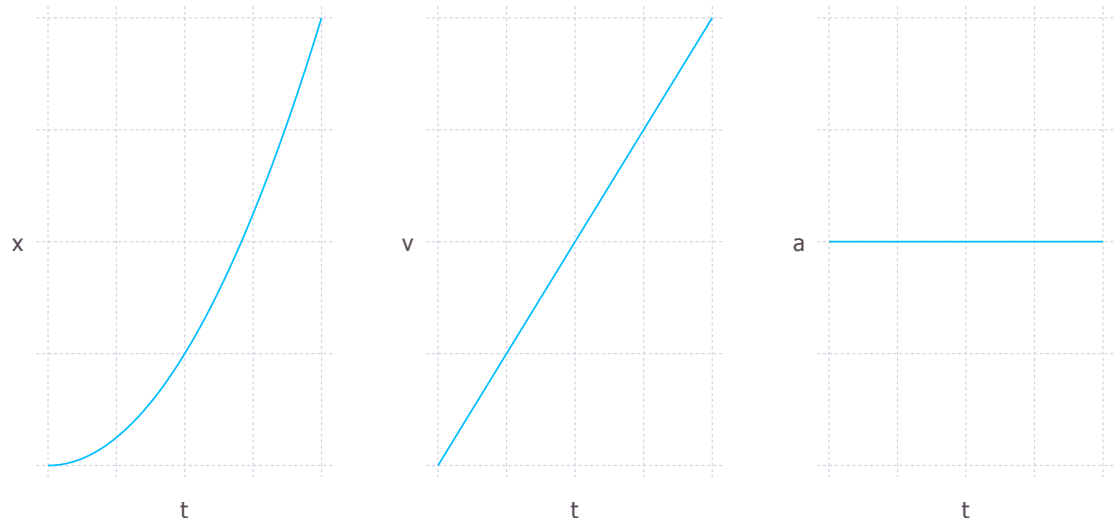
Za definiranje varijable *zsurf* koristili smo operatore koji počinju s točkom: .* i .^. Oni ne množe cijele nizove ili matrice, nego samo elemente odgovarajućih indeksa. Uz navedene funkcije, iskorištene su i funkcije koje rotiraju trodimenzionalni graf u položaj koji je pregledniji.

```
figure()
ax = gca(projection="3d")
plot_surface(xpot,ypot,zpot,cmap=ColorMap("coolwarm"))
xlabel("x")
ylabel("y")
zlabel("U(x,y)")
ax[:view_init](45, 40)
plot3D(xtraj,ytraj,zsurf,color="#990000",linewidth=2)
savefig("cestupot3d.png")
```

6 Metodički dio: Određivanje kinematičkih veličina numeričkim metodama - učenički projekt

6.1 Uvod

Kinematičke veličine kao što su položaj, brzina i akceleracija se uče u prvom razredu srednje škole. Uz njih se uče i grafovi koji ih opisuju. Učeničke poteškoće kod razumijevanja grafova su česte, a i potrebno predznanje informatike u nižim razredima srednje škole je još uvijek slabo, tako da bi učenički projekt koji se bavi određivanjem kinematičkih veličina numeričkim metodama bio prikladniji za učenike trećeg ili četvrtog razreda srednjih škola s četverogodišnjim programima fizike i informatike.



Slika 6.1: Primjeri *x*-*t*, *v*-*t* i *a*-*t* grafa za gibanje s konstantnom akceleracijom

Grafovi kinematičkih veličina s kojima učenici imaju poteškoća su: *x*-*t* (položaj u ovisnosti o vremenu), *v*-*t* (brzina u ovisnosti o vremenu) i *a*-*t* graf (akceleracija u ovisnosti o vremenu). Nagib grafa položaja u ovisnosti o vremenu predstavlja brzinu, dok nagib grafa brzine u ovisnosti o vremenu predstavlja akceleraciju.

$$v = \frac{\Delta x}{\Delta t} \tag{6.1}$$

$$a = \frac{\Delta v}{\Delta t} \tag{6.2}$$

Isto tako površina ispod grafa akceleracije u ovisnosti o vremenu predstavlja promjenu brzine, dok površina ispod grafa brzine u ovisnosti o vremenu predstavlja prijeđeni put.

$$\Delta v = a\Delta t \tag{6.3}$$

$$\Delta x = v\Delta t \tag{6.4}$$

Neke od najčešćih učeničkih poteškoća vezano uz grafove su: graf kao slika putanje, zamjena visine i nagiba grafa, zamjena varijabli, greške kod grafova koji ne prolaze ishodištem, nepoznavanje značenja površine ispod grafa i nemogućnost izračuna površine ispod grafa, zamjena intervala točkom. [5, 6]

Poteškoća grafa kao slika putanje se javlja kada se graf koji je simbolička veza među varijablama percipira kao fotografija gibanja. Učenici od kojih se traži da nacrtaju graf brzine u ovisnosti o vremenu automobila koji se kreće po brdovitoj cesti često crtaju grafove koji izgledaju kao brda. Također česta poteškoća je zamjena visine i nagiba grafa. Učenici koji imaju ovu poteškoću umjesto mjesta s najvećim nagibom, kao mjesto najveće promjene, prepoznaju najveću vrijednost na grafu, ili najveću razliku vrijednosti. Zamjena varijabli znači da učenici ne razlikuju položaj, brzinu i akceleraciju. Takva poteškoća se očituje time što učenik mijenja osi grafa, a da pri tome ne shvaća da se i ovisnost na grafu mora mijenjati. Uz to, probleme učenicima zadaju i nagibi linija koje ne prolaze kroz ishodište. Probleme s izračunima imaju i učenici koji u čistom matematičkom kontekstu znaju izračunati nagib i površinu ispod grafa, ali ne i u fizikalnom kontekstu. [5, 6]

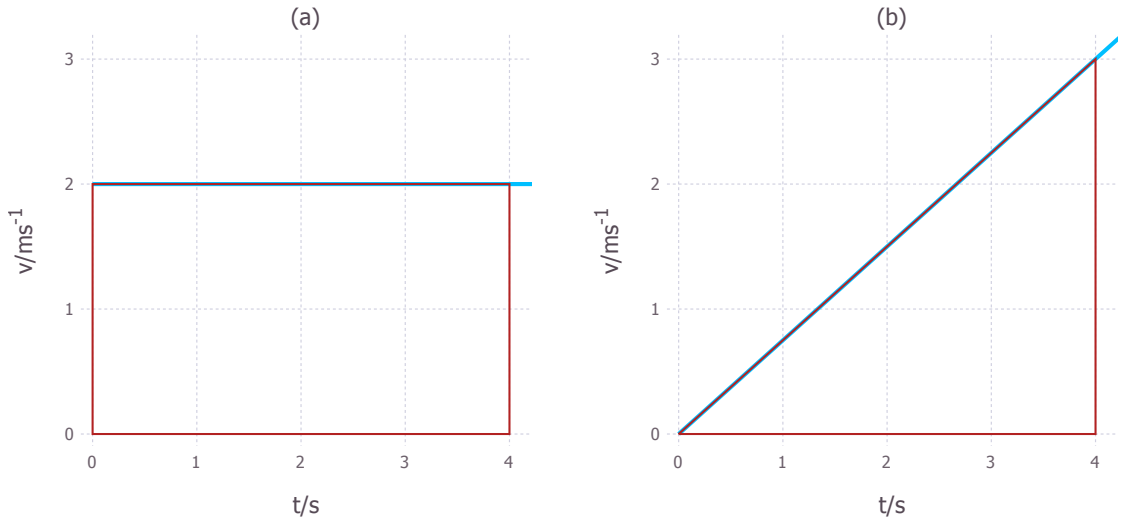
Projekt određivanja kinematičkih veličina numeričkim metodama se sastoji od dvije komponente: razumijevanje grafova kinematičkih veličina i primjena tog znanja i znanja iz informatike na problem računanja površine ispod grafa, radi određivanja kinematičke veličine. Moguće je numeričko određivanje nagiba grafa, to jest derivacije funkcije, ali numerički algoritmi za deriviranje su preosjetljivi na numeričke greške da bi bili pogodni za učenički projekt.

Učenički projekt bio bi zadan na početku polugodišta, a završetak projekta bio bi nakon četiri tjedna. Na početku učeniku treba objasniti problematiku projekta, zatim mu dati vremena za samostalno istraživanje problematike i nakon otprilike dva tjedna provjeriti učenikovo razumijevanje problema i status projekta. Nakon toga treba održavati sastanke prema potrebi. Kao rezultat projekta od učenika se očekuje referat i desetominutno izlaganje pred razredom. Sadržaj referata i izlaganja bi trebao uključiti opis x-t, v-t i a-t grafova, objašnjenje značenja površine ispod grafa, računanje površine ispod grafa za pravilne grafove (površine ispod grafa se mogu svesti na jednostavne geometrijske likove) i algoritam, ili program, koji računa površinu ispod proizvoljne krivulje a-t ili v-t grafa primijenjen na zadatku s fizikalnim kontekstom.

Projekt koji bi učenici samostalno radili, a koji zahtjeva korištenje znanja stečenoga o kinematičkim veličinama na nastavi u kontekstu novih znanja koje stječu kroz algoritme za numeričko računanje istih tih veličina, pomogao bi u otklanjanju ovakvih poteškoća. Također je važna spoznaja da znanja stečena na drugim predmetima (poput informatike) mogu koristiti u kontekstu fizike, kao i obrnuto. Veća povezanost među raznim predmetima može doprinijeti većoj motivaciji u radu (ako znanja iz ”najdražeg” predmeta koriste u ”ne tako dragim” predmetima) i shvaćanju da ono što nauče ima i primjenu u nekim drugim poljima, što može poticati kreativnost.

6.2 *Razvijanje ideje integriranja*

Osnovna ideja integracije je zbrajanje velikog broja malih površina (zapravo beskonačnog broja infinitezimalnih površina).



Slika 6.2: Primjeri grafova čije su površine geometrijski likovi: (a) pravokutnik i (b) trokut

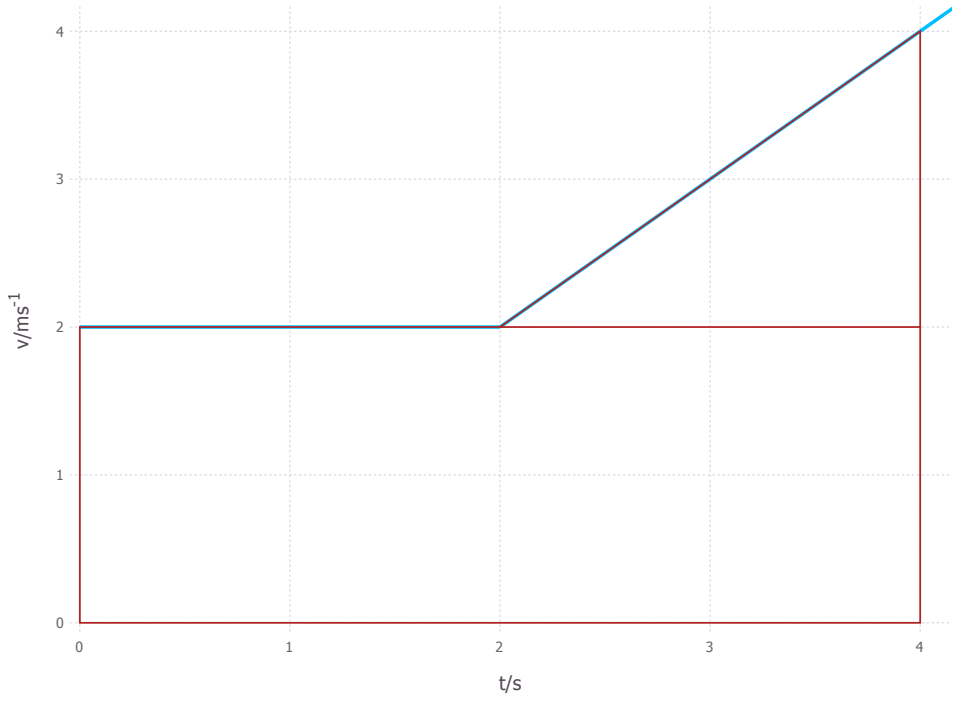
Grafovima, kao na slici 6.2, je lagano izračunati površinu. Za pravokutnik se očitaju duljine stranica s osi grafa i zatim se pomnože. Kod grafova koji su pravokutnik je ovaj postupak lakše za shvatiti nego za druge oblike grafova. Ali drugi oblici grafova se mogu svesti na velik broj površina koje su pravokutnici, gdje je visina pravokutnika vrijednost funkcije na grafu, a dužina neki prikladni segment apscise. Prije nego što numeričke metode bile dovoljno razvijene, zbog tehnoloških razloga, ovakva metoda se koristila za računanje površina ispod grafova, ali su elementi bili kvadrati. Konkretno, zbrajali su se kvadratići na milimetarskom papiru. Učeniku je bitno demonstrirati, to jest pitanjima ga dovesti do zaključka da je broj ovakvih površina jednak površini ispod grafa. Prvo treba provjeriti prihvaća li učenik činjenicu da je površina pravokutnika ispod grafa kao u primjeru (a) na slici 6.2 vrijednost prijeđenog puta. Ako je tako, onda ga treba pitati može li se primjer (b) raščlaniti na više segmenata; što ako gledamo male vremenske intervale? Kada ga se u to uvjeri, treba pitati kava bi bila brzina u takvim malim vremenskim intervalima. Da li se brzina puno promjeni u malom intervalu? Da li se može pretpostaviti da je stalna? Kojeg je tada oblika segment koji čine interval vremena i visina grafa brzine? Zadnji korak bi činilo osvještavanje činjenice da se sve te segmente može zbrojiti, što za dovoljno male segmente daje površinu ispod grafa. Treba pitati, što ako zbrojimo sve male intervale vremena? Ako se ovakav koncept shvati i osnovnoškolske formule za površinu su primjenjive, na primjer za trokut se očitaju duljine kateta, pomnože

se i rezultat se podijeli s dva.

$$\Delta x = P_a = 2 \cdot 4 = 8\text{m} \tag{6.5}$$

$$\Delta x = P_b = \frac{4 \cdot 3}{2} = 6\text{m} \tag{6.6}$$

Čak se i neki malo kompliciraniji grafovi, kao na slici 6.3, mogu svesti na kombinaciju geometrijskih likova. Prvo se izračuna površina pravokutnika, zatim trokuta i onda



Slika 6.3: Primjeri čija se površina svodi na pravokutnik i trokut

se zbroje.

$$\Delta x = P_{pravokutnik} + P_{trokut} = 2 \cdot 4 + \frac{(4-2) \cdot (4-2)}{2} = 10\text{m} \tag{6.7}$$

Ovakva metoda je jednostavna i prilično brza, ali nije ju moguće provesti za proizvoljnu funkciju koja predstavlja brzinu ili akcelaraciju, da bi se dobila promjena položaja, odnosno brzine.

Uz to bitno je da učenik shvati da je površina ispod grafa, koji je simbolički prikaz ovisnosti fizikalnih veličina, također predstavlja vrijednost fizikalne veličine. Jednadžbe u uvodnom dijelu pokazuju da množenjem dvije veličine dobivamo treću, isto tako množenjem dvije duljine dobiva se površina. Ako su te duljine simbolička reprezentacija fizikalnih veličina onda je i njihov umnožak fizikalna veličina. Znači da je i površina ispod grafa fizikalne veličine također reprezentacija fizikalne veličine.

$$\frac{[m]}{[s]^2} \cdot [s] = \frac{[m]}{[s]} \tag{6.8}$$

6.3 Numerička integracija metodom pravokutnika

Korištenjem numeričke integracije moguće je izračunati površinu ispod grafa za proizvoljnu funkciju. Metoda koja se koristi nije mnogo drugačija od metode za izračun površine grafova koje su geometrijski oblici. U ovom slučaju se površina svodi na velik broj pravokutnika. Apscisa se podijeli na pravilne segmente u intervalu u kojem se želi integrirati. Za početak svakog tog segmenta funkcija se izvrijedni, to jest odredi visina grafa. Kada se te dvije vrijednosti pomnože dobije se površina pravokutnika kojemu je jedna stranica duljina segmenta, a druga vrijednost funkcije na početku segmenta. Kada se sve to ponovi za sljedeći segment dobiva se još jedna površina. Nakon što se dobiju površine svih takvih pravokutnika, zbroje se da bi se dobila ukupna površina. Algoritam za ovaj postupak je jednostavan.

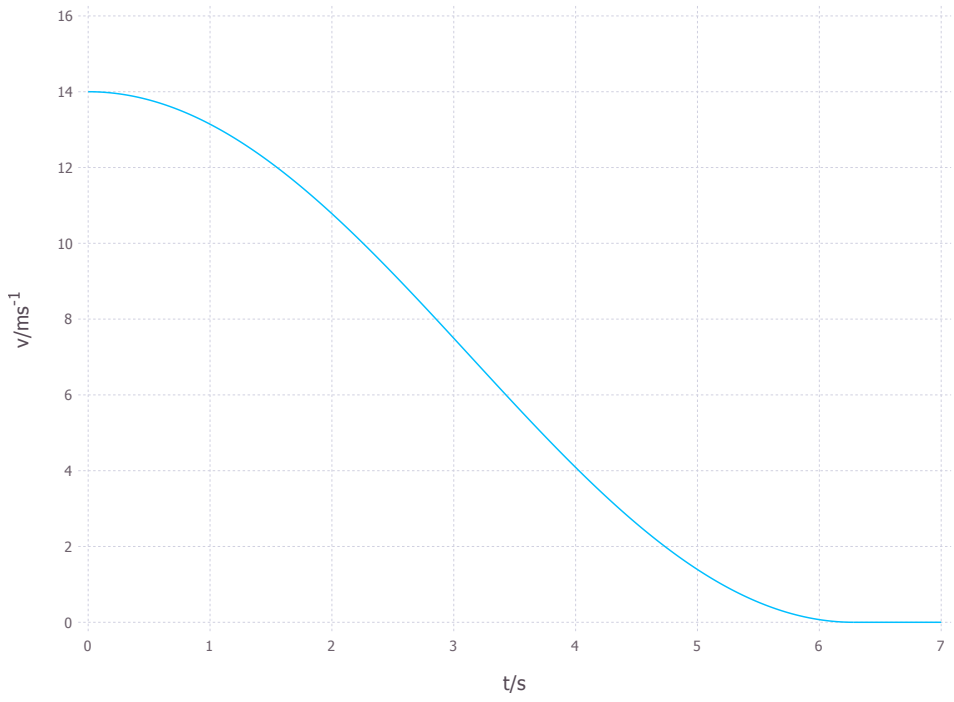
```
function povrsina(f,a,b;inc=0.0001)
    s=0.0
    for x=a:inc:b
        s+=f(x)*inc
    end
    return s
end
```

Definira se funkcija imena *povrsina* koja kao argument prima funkciju koju treba integrirati (*f*) i granice integracije (*a* i *b*). Kao argument s ključnom riječi (*inc*) se definira veličina segmenta. O veličini segmenta ovisit će točnost izračuna, što je segment manji rezultat je točniji. Na početku funkcije definira se varijabla *s*, postavljanjem na nulu. Nakon toga *for* petlja u intervalu od *a* do *b* veličine segmenta *inc* zbraja površine, visine vrijednosti funkcije i duljine segmenta, u varijablu *s*. Funkcija *povrsina()* zatim vraća vrijednost varijable *s* kao rezultat.

Ovako napisan algoritam je veoma sažet. Od učenika se ne očekuje ovakav rezultat. Algoritam je moguće napisati tako da nije dio funkcije ili da koristi *while* petlju umjesto *for* petlje. Također, iako je ovaj algoritam pisan u Juliji, od učenika se ne očekuje učenje novog programskog jezika, već korištenje onoga koji poznaje. Učenički algoritam treba imati sljedeće komponente: računa površine za svaki pojedini segment vremena, zbraja sve površine i daje valjan rezultat u bilo kojem obliku (ispis niza znakova, spremanje u varijablu ili spremanje u datoteku).

6.4 Zadatak

Automobil se kreće nekom brzinom. Vozač odluči kočiti. Nakon 6.2 sekunda automobil se potpuno zaustavi. Ovakvo gibanje je opisano jednadžbom $v(t) = 7(1+cos(t/2))$ i *v-t* grafom priloženom zadatku.



Slika 6.4: Graf kočenje automobila (prilog zadatku)

- Pomoću grafa odredi brzinu kojom se automobil kretao prije nego što je počeo kočiti?
- Opiši gibanje iz zadatka! Kakva je akceleracija tijekom gibanja? Obrazloži!
- Procijeni, i zatim koristeći numeričke metode izračunaj koliki je zaustavni put automobila!

Očekivani odgovori na ovaj zadatak bi morali biti sljedećeg oblika. Prvo, vrijednost brzine u trenutku 0 je iznosila $14\,\mathrm{m/s}$. Drugo, od 0 do otprilike 1 sekunde automobil lagano usporava, od 1 do 5 sekundi značajno usporava. S obzirom da nagib na *v*-*t* grafu predstavlja akceleraciju, ovo je vidljivo iz nagiba grafa koji je manji u prvom dijelu, a veći u drugom. Također je nagib opet manji u zadnjem dijelu, što odgovara laganijem usporavanju. Također bi bilo dobro da, osim opisa, učenik i izračuna akceleraciju pomoću grafa za sva tri područja. Odgovor na treće potpitanje mora sadržavati procjenu na temelju grafa i valjan izračun, koji za program dan prije iznosi otprilike 44 m, te kod i objašnjenje programa kojim je izvršen izračun. Učenički program mora zadovoljavati prije navedene uvjete.

7 Zaključak

U ovom radu korišten je programski jezik Julija i razmatrane su njegove moguće primjene u znanosti, kroz rješavanje fizikalnih problema numeričkim putem. Julija se pokazala kao programski jezik jednostavan za učenje onima koji već poznaju jezike kao što su Python i Matlab. Brzina Julije je usporediva s C-om za pravilno pisane programe.

Veliki broj paketa omogućava rješavanje velikog broja problema, ali su ti paketi, kao i sama Julija u ranoj fazi razvoja (verzija Julije je 0.4.6 u trenutku pisanja rada), te su često održavani od strane samo jedne osobe i dosta slabo dokumentirani. Također velik broj paketa oslanja se na vanjske biblioteke, najčeće Pythonove, što zahtjeva i njegovu instalaciju, kao i instalaciju potrebnih biblioteka.

Usprkos nedostacima Julija je moćan alat, kojim su problemi izneseni u ovom radu riješeni u svega par linija koda. Daljnjim razvojem će sigurno postati još moćnija.

Literatura

[1] Balbaert, I. Getting Started with Julia Programming. Birmingham : Packt Publishing Ltd, 2015.

[2] Sherrington, M. Mastering Julia. Birmingham : Packt Publishing Ltd, 2015.

[3] Hjorth-Jensen, M. Computational Physics. Oslo : University of Oslo, 2009.

[4] Julia.org, `http://julialang.org/`, 6.9.2016.

[5] Beichnrt, R. J. Testing student interpretation of kinematics graphs. // American journal of Physics, Vol. 62.8(1994), str. 750-762

[6] Planinić, M.; Ivanjek, L.; Sušac, A.; Milin-Šipuš, Ž. Comparison of university students' understanding of graphs in different contexts. // Physical Review Special Topics - Physics Education Research, Vol. 9.2(2013)